Complete Code Figures from
“Designing Audio Objects” by Eric Lyon

```
outlets = 2;

function buildsine()
{
   var length = 1024;
   var i;
   var sineval;
   for(i = 0; i < length; i++){
         sineval = Math.sin(2 * Math.PI * i / length);
         outlet(1,sineval); // set the value
         outlet(0,i); // send index trigger buffer write
   }
}
```

Figure 2.6 JavaScript code to generate a sine wave

```
for(i = 0; i < sample_count, i++) {
  output[i] = input1[i] * input2[i];
}
```

Figure 3.1 A C `for` loop to multiply the contents of two arrays

```
while(1){
  readsamp(&input1, &input2); // hypothetical read function
  output = input1 * input2; // do the multiplication
  writesamp(&output); // hypothetical write function
}
```

Figure 3.2 An infinite `while` loop for multiplying two input sample streams

```
for(i = 0; i < n; i++){
  out[i] = in1[i] * in2[i];
}
```

Figure 3.3 Processing Max signal vectors with a signal vector size of `n` samples

```
while(n--){
  *out++ = *in1++ * *in2++;
}
```

Figure 3.4 Processing Max signal vectors using pointer arithmetic

```
#include "ext.h"
#include "z_dsp.h"
#include "ext_obex.h"
```

Figure 3.5 Required header files for Max/MSP audio DSP objects

```
typedef struct _multy {
  t_pxobject obj;
} t_multy;
```

Figure 3.6 The Max/MSP object structure

```
struct multy { t_pxobject obj; };
typedef struct _multy t_multy;
```

Figure 3.7 The object structure from figure 3.6 with structure and type defined separately

```
typedef struct {
  t_pxobject obj;
} t_multy;
```

Figure 3.8 A more compact combined object structure and type definition

```
static t_class *multy_class;
```

Figure 3.9 The *multy~* class pointer

```
void *multy_new(void);
void multy_dsp(t_multy *x, t_signal **sp, short *count);
t_int *multy_perform(t_int *w);
```

Figure 3.10 Function prototypes

```
void *multy_new(void);
```

Figure 3.11 The first line of the `multy_new()` function.

```
int main(void)
{
   multy_class = class_new("multy~", (method)multy_new,
         (method)dsp_free, sizeof(t_multy), 0,0);
   class_addmethod(multy_class,(method)multy_dsp, "dsp",A_CANT,0);
   class_dspinit(multy_class);
   class_register(CLASS_BOX, multy_class);
   post("multy~ from \"Designing Audio Objects\" by Eric Lyon");
   return 0;
}
```

Figure 3.12 The initialization routine

```
void *multy_new(void)
{
    t_multy *x = (t_multy *)object_alloc(multy_class);
    dsp_setup((t_pxobject *)x, 2);
    outlet_new((t_object *)x, "signal");
    return x;
}
```

Figure 3.13 The new instance routine.

```
void multy_dsp(t_multy *x, t_signal **sp, short *count)
{
    dsp_add(multy_perform, 5, x, sp[0]->s_vec, sp[1]->s_vec,
          sp[2]->s_vec, sp[0]->s_n);
}
```

Figure 3.14 The *multy~* dsp method.

```
void multy_dsp(t_multy *x, t_signal **sp, short *count)
{
  dsp_add(multy_perform, 5, x, sp[0]->s_vec-1,
    sp[1]->s_vec-1, sp[2]->s_vec-1, sp[0]->s_n+1);
}
```

Figure 3.15 A pre-decrement/increment form of the dsp method

```
while(--n){
  *++out = *++in1 * *++in2;
}
```

Figure 3.16 The perform loop in pre-decrement/increment processing

```
t_int *multy_perform(t_int *w)
{
   t_multy *x = (t_multy *) (w[1]);
   t_float *in1 = (t_float *) (w[2]);
   t_float *in2 = (t_float *) (w[3]);
   t_float *out = (t_float *) (w[4]);
   t_int n = w[5];

   while(n--){
         *out++ = *in1++ * *in2++;
   }
   return w + 6;
}
```

Figure 3.17 The *multy~* perform routine

```
void multy_assist(t_multy *x, void *b, long msg, long arg,
      char *dst)
{
  if (msg==ASSIST_INLET) {
    switch (arg) {
    case 0:
      sprintf(dst,"(signal) Input 1");
      break;
    case 1:
      sprintf(dst,"(signal) Input 2");
      break;
    }
  }
  else if (msg==ASSIST_OUTLET) {
    sprintf(dst,"(signal) Output");
  }
}
```

Figure 3.19 The assist method.

```
void multy_assist(t_multy *x, void *b, long msg, long arg,
      char *dst);
```

Figure 3.20 The assist function prototype

```
class_addmethod(multy_class,(method)multy_assist, "assist",
      A_CANT, 0);
```

Figure 3.21 Binding the assist method to the *multy~* class

```
#include "m_pd.h"
```

Figure 3.24 The required Pd header

```
static t_class *multy_class;
```

Figure 3.25 The Pd *multy~* class pointer

```
typedef struct _multy
{
     t_object obj;
     t_float x_f;
} t_multy;
```

Figure 3.26 The Pd object structure

```
void *multy_new(void);
void multy_dsp(t_multy *x, t_signal **sp, short *count);
t_int *multy_perform(t_int *w);
```

Figure 3.27 The Pd function prototypes

```
void multy_tilde_setup (void)
{
    multy_class = class_new(gensym("multy~"),
          (t_newmethod)multy_new, 0, sizeof(t_multy), 0, 0);
    CLASS_MAINSIGNALIN(multy_class, t_multy, x_f);
    class_addmethod(multy_class, (t_method)multy_dsp,
          gensym("dsp"),0);
    post("multy~ from \"Designing Audio Objects\" by Eric Lyon");
}
```

Figure 3.28 The Pd class definition function

```
void *multy_new( void )
{
    t_multy *x = (t_multy *) pd_new(multy_class);
    inlet_new(&x->obj, &x->obj.ob_pd, gensym("signal"),
          gensym("signal"));
    outlet_new(&x->obj, gensym("signal"));
    return x;
}
```

Figure 3.29 The Pd new instance routine for *multy~*

```
float my_array[64];
```

Figure 4.2 Static memory allocation

```
float max_ms = 250;
int srate = 44100;
int delay_length;
delay_length = max_ms * 0.001 * (float) srate;
delay_length += 1;
```

Figure 4.3 Calculating the size of the memory in samples

```
float *delay_line;
```

Figure 4.4 The delay line, defined as a pointer to floats

```
delay_line = (float *) sysmem_newptr(100000 * sizeof(float));
```

Figure 4.5 Allocating dynamic memory for the delay line

```
float delay_line[100000]; // cannot use in Max object structure
```

Figure 4.6 Static memory allocation for the delay line

```
float *delay_line;
delay_bytes = delay_length * sizeof(float);
delay_line = (float *) sysmem_newptr(delay_bytes);
```

Figure 4.7 Dynamic memory allocation based on the desired size of the delay line

```
int write_index = 0; // initialized outside perform method
// *input is the MSP buffer containing the signal to be delayed
while(n--){
  // write sample into next available location and increment index
  delay_line[write_index++] = *input;
  // keep within range of the array length
  if( write_index >= delay_length){
    write_index -= delay_length;
  }
}
```

Figure 4.8 Writing to a circular buffer

```
while(n--){
  *output++ = delay_line[read_index - 4410];
  delay_line[read_index++] = *input++;
  if( read_index >= delay_length){
    read_index -= delay_length;
  }
}
```

Figure 4.9 First attempt at reading from a circular buffer

```
read_index = write_index - 4410;
while(read_index < 0)
  read_index += delay_length;
```

Figure 4.10 Avoiding illegal indexing

```
while(n--){
  *output++ = *input++ * gain;
}
```

Figure 4.11 Typical DSP code where buffer sharing works correctly

```
out_sample = delay_line[read_index++];
delay_line[write_index++] = *input++;
*output++ = out_sample;
```

Figure 4.12 Storing the input sample before writing over it

```
dsp_setup((t_pxobject *)x, 3); //initialize object with 3 inlets
x->obj.z_misc |= Z_NO_INPLACE; //force independent signal vectors
```

Figure 4.13 Setting the Max/MSP flag to prohibit buffer sharing

```
int idelay;
idelay = round(ms_delay * sr * 0.001);
```

Figure 4.14 Truncating the delay time

```
void vdelay_dsp(t_vdelay *x, t_signal **sp, short *count)
{
  dsp_add(vdelay_perform, 6, x, sp[0]->s_vec, sp[1]->s_vec,
    sp[2]->s_vec, sp[3]->s_vec, sp[0]->s_n);
}
```

Figure 4.15 The dsp method for *vdelay~*

```
typedef struct _vdelay {
  t_pxobject obj;
  float sr; // sampling rate
  float maximum_delay_time; // maximum delay time
  long delay_length; // length of the delay line in samples
  long delay_bytes; // length of delay line in bytes
  float *delay_line; // the delay line itself
  float delay_time; // current delay time
  float feedback; // feedback multiplier
  long write_index; // write point in delay line
  long read_index; // read point in delay line
  short delaytime_connected; // inlet connection status
  short feedback_connected; // inlet connection status
} t_vdelay;
```

Figure 4.16 The object structure for *vdelay~*

```
void *vdelay_new(t_symbol *s, short argc, t_atom *argv)
{
   int i;
   float delmax = 100.0, deltime = 100.0, feedback = 0.1;
   t_vdelay *x = object_alloc(vdelay_class);
   dsp_setup((t_pxobject *)x, 3);
   outlet_new((t_object *)x, "signal");
   x->obj.z_misc |= Z_NO_INPLACE;
   x->sr = sys_getsr();

   atom_arg_getfloat(&delmax, 0, argc, argv);
   atom_arg_getfloat(&deltime, 1, argc, argv);
   atom_arg_getfloat(&feedback, 2, argc, argv);

   if(delmax <= 0){
         delmax = 250.0;
   }
   x->maximum_delay_time = delmax * 0.001;

   x->delay_time = deltime;
   if(x->delay_time > delmax || x->delay_time <= 0.0){
         error("vdelay~: illegal delay time: %f", x->delay_time);
         x->delay_time = 1.0;
   }
   x->delay_length = x->sr * x->maximum_delay_time + 1;
   x->delay_bytes = x->delay_length * sizeof(float);
   x->delay_line = (float *) sysmem_newptr(x->delay_bytes);
   if(x->delay_line == NULL){
         error("vdelay~: cannot allocate %d bytes of memory",
               x->delay_bytes);
         return NULL;
   }
   for(i = 0; i < x->delay_length; i++){
         x->delay_line[i] = 0.0;
   }
   x->feedback = feedback;
   x->write_index = 0;
   return x;
}
```

Figure 4.17 The new instance routine for *vdelay~*

```
dsp_setup(&x->obj, 3);
```

Figure 4.18 Passing a pointer to the proxy object component without casting

```
t_int *vdelay_perform(t_int *w)
{
   t_vdelay *x = (t_vdelay *) (w[1]);
   t_float *input = (t_float *) (w[2]);
   t_float *delaytime = (t_float *) (w[3]);
   t_float *feedback = (t_float *) (w[4]);
   t_float *output = (t_float *) (w[5]);
   t_int n = w[6];
   float sr = x->sr;
   float *delay_line = x->delay_line;
   long read_index = x->read_index;
   long write_index = x->write_index;
   long delay_length = x->delay_length;
   long idelay;
   float srms = sr / 1000.0;
   float out_sample;

   while(n--){
         idelay = round(*delaytime++ * srms);
         if(idelay < 0){
               idelay = 0;
         }
         else if(idelay > delay_length){
               idelay = delay_length - 1;
         }
         read_index = write_index - idelay;
         while(read_index < 0){
               read_index += delay_length;
         }
         out_sample = delay_line[read_index];
         delay_line[write_index++] =
               *input++ + out_sample * *feedback++;
         *output++ = out_sample;
         if(write_index >= delay_length){
               write_index -= delay_length;
         }
   }
   x->write_index = write_index;
   return w + 7;
}
```

Figure 4.19 The perform routine for *vdelay~*

```
void vdelay_assist(t_vdelay *x, void *b, long msg, long arg,
   char *dst)
{
   if (msg == ASSIST_INLET) {
         switch (arg) {
               case 0: sprintf(dst,"(signal) Input"); break;
               case 1: sprintf(dst,"(signal) Delay Time"); break;
               case 2: sprintf(dst,"(signal) Feedback"); break;
         }
   }
   else if (msg == ASSIST_OUTLET) {
         sprintf(dst,"(signal) Output");
   }
}
```

Figure 4.20 The assist method for *vdelay~*

```
void vdelay_free(t_vdelay *x)
{
   dsp_free((t_pxobject *) x);
   sysmem_freeptr(x->delay_line);
}
```

Figure 4.21 The free function for *vdelay~*

```
int main(void)
{
   vdelay_class = class_new("vdelay~",
         (method)vdelay_new,(method)vdelay_free,
         sizeof(t_vdelay),0,A_GIMME,0);
   class_addmethod(vdelay_class, (method)vdelay_dsp, "dsp",
         A_CANT, 0);
   class_addmethod(vdelay_class, (method)vdelay_assist, "assist",
         A_CANT, 0);
   class_dspinit(vdelay_class);
   class_register(CLASS_BOX, vdelay_class);
   post("vdelay~ from \”Designing Audio Objects\” by Eric Lyon");
   return 0;
}
```

Figure 4.22 The initialization routine for *vdelay~*

```
void vdelay_dsp(t_vdelay *x, t_signal **sp, short *count)
```

```
{
   int i;
   if(x->sr != sp[0]->s_sr){
         x->sr = sp[0]->s_sr;
         x->delay_length = x->sr * x->maximum_delay_time + 1;
         x->delay_bytes = x->delay_length * sizeof(float);
         x->delay_line =
               (float *) sysmem_resizeptr((void *)x->delay_line,
               x->delay_bytes);
         if(x->delay_line == NULL){
               error("vdelay~: cannot realloc %d bytes of memory",
                     x->delay_bytes);
               return;
         }
         for(i = 0; i < x->delay_length; i++){
               x->delay_line[i] = 0.0;
         }
         x->write_index = 0;
   }
   dsp_add(vdelay_perform, 6, x, sp[0]->s_vec, sp[1]->s_vec,
         sp[2]->s_vec, sp[3]->s_vec, sp[0]->s_n);
}
```

Figure 4.24 The revised dsp method for *vdelay~*

```
void vdelay_float(t_vdelay *x, double f)
{
   int inlet = ((t_pxobject*)x)->z_in;
   switch(inlet){
         case 1: // 2nd inlet
               if(f < 0.0 ||
                  f > x->maximum_delay_time * 1000.0){
                     error("vdelay~: illegal delay: %f", f);
               } else {
                     x->delay_time = f;
               }
               break;
         case 2: // 3rd inlet
               x->feedback = f;
               break;
   }
}
```

Figure 4.25 The float method for *vdelay~*

```
class_addmethod(vdelay_class,(method)vdelay_float,"float",A_FLOAT,0);
```

Figure 4.26 Binding the float method

```
x->delaytime_connected = count[1];
```

```
x->feedback_connected = count[2];
```

Figure 4.27 Storing the connection states of the rightmost two inlets

```
t_int *vdelay_perform(t_int *w)
{
    t_vdelay *x = (t_vdelay *) (w[1]);
    t_float *input = (t_float *) (w[2]);
    t_float *delaytime = (t_float *) (w[3]);
    t_float *feedback = (t_float *) (w[4]);
    t_float *output = (t_float *) (w[5]);
    t_int n = w[6];
    float sr = x->sr;
    float *delay_line = x->delay_line;
    long read_index = x->read_index;
    long write_index = x->write_index;
    long delay_length = x->delay_length;
    short delaytime_connected = x->delaytime_connected;
    short feedback_connected = x->feedback_connected;
    float delaytime_float = x->delay_time;
    float feedback_float = x->feedback;
    long idelay;
    float srms = sr / 1000.0;
    float out_sample;

    while(n--){
        if(delaytime_connected){
            idelay = round(*delaytime++ * srms);
        }
        else {
            idelay = round(delaytime_float * srms);
        }
        if(idelay < 0){
            idelay = 0;
        }
        else if( idelay > delay_length){
            idelay = delay_length - 1;
        }
        read_index = write_index - idelay;
        while(read_index < 0){
            read_index += delay_length;
        }
        out_sample = delay_line[read_index];
        if(feedback_connected) {
            delay_line[write_index++] = *input++
                + out_sample * *feedback++;
        }
        else {
            delay_line[write_index++] = *input++
                + out_sample * feedback_float;
        }
        *output++ = out_sample;
        if(write_index >= delay_length){
            write_index -= delay_length;
        }
    }
    x->write_index = write_index;
    return w + 7;
}
```

Figure 4.28 The *vdelay~* perform routine modified to accept float or signal input

```
sprintf(dest, "(signal/float) Delay Time");
```

Figure 4.29 Assist string indicating that either float or signal input is accepted

```
fraction = delaytime - trunc(delaytime);
m1 = 1. - fraction;
m2 = fraction;
interp_sample = m1 * samp1 + m2 * samp2;
```

Figure 4.30 Code to implement linear interpolation

```
interp_sample = samp1 + fraction * (samp2 - samp1);
```

Figure 4.31 A more efficient calculation of linear interpolation

```
t_int *vdelay_perform(t_int *w)
{
    t_vdelay *x = (t_vdelay *) (w[1]);
    t_float *input = (t_float *) (w[2]);
    t_float *delaytime = (t_float *) (w[3]);
    t_float *feedback = (t_float *) (w[4]);
    t_float *output = (t_float *) (w[5]);
    t_int n = w[6];
    float sr = x->sr;
    float *delay_line = x->delay_line;
    long read_index = x->read_index;
    long write_index = x->write_index;
    long delay_length = x->delay_length;
    short delaytime_connected = x->delaytime_connected;
    short feedback_connected = x->feedback_connected;
    float delaytime_float = x->delay_time;
    float feedback_float = x->feedback;
    float fraction;
    float fdelay;
    float samp1, samp2;
    long idelay;
    float srms = sr / 1000.0;
    float out_sample;

    while(n--){
        if(delaytime_connected){
            fdelay = *delaytime++ * srms;
        }
        else {
            fdelay = delaytime_float * srms;
        }
        while(fdelay < 0){
            fdelay += delay_length;
        }
        idelay = trunc(fdelay);
        fraction = fdelay - idelay;
        read_index = write_index - idelay;
        while(read_index < 0){
            read_index += delay_length;
        }
        samp1 = delay_line[read_index];
        samp2 = delay_line[(read_index + 1) % delay_length];
        out_sample = samp1 + fraction * (samp2-samp1);
        if(feedback_connected) {
            delay_line[write_index++] = *input++
                + out_sample * *feedback++;
        }
        else {
            delay_line[write_index++] = *input++
                + out_sample * feedback_float;
        }
        *output++ = out_sample;
        if(write_index >= delay_length){
            write_index -= delay_length;
        }
    }
    x->write_index = write_index;
    return w + 7;
}
```

Figure 4.32 The *vdelay~* perform routine with delay-line interpolation

```
#include "m_pd.h"
#include "math.h"
```

Figure 4.33 Required header files for the Pd version of *vdelay~*

```
t_object obj; // t_object rather than t_pxobject
t_float x_f; // convert float to signal
```

Figure 4.34 Slight changes to the Pd object structure

```
void vdelay_tilde_setup (void)
{
    vdelay_class =

    class_new(gensym("vdelay~"),(t_newmethod)vdelay_new,
          (t_method)vdelay_free,sizeof(t_vdelay),0,A_GIMME,0);
    CLASS_MAINSIGNALIN(vdelay_class, t_vdelay, x_f);
    class_addmethod(vdelay_class, (t_method)vdelay_dsp,
                gensym("dsp"), A_CANT, 0);
    post("vdelay~: from Designing Audio Objects (Pd version)");
}
```

Figure 4.35 The class definition routine for Pd version of *vdelay~*

```
void *vdelay_new(t_symbol *s, short argc, t_atom *argv)
{
   int i;
   float delmax = 100.0, deltime = 100.0, feedback = 0.1;
   t_vdelay *x = (t_vdelay *) pd_new(vdelay_class);
   inlet_new(&x->obj, &x->obj.ob_pd, gensym("signal"),
         gensym("signal"));
   inlet_new(&x->obj, &x->obj.ob_pd, gensym("signal"),
         gensym("signal"));
    outlet_new(&x->obj, gensym("signal"));
   x->sr = sys_getsr();
   if(argc >= 3){ feedback = atom_getfloatarg(2, argc, argv); }
   if(argc >= 2){ deltime = atom_getfloatarg(1, argc, argv); }
   if(argc >= 1){ delmax = atom_getfloatarg(0, argc, argv); }
   if(delmax <= 0){
         delmax = 250.0;
   }
   x->maximum_delay_time = delmax * 0.001;
   x->delay_time = deltime;
   if(x->delay_time > delmax || x->delay_time <= 0.0){
         error("bad delay time: %f, reset to 1 ms",
               x->delay_time);
         x->delay_time = 1.0;
   }
   x->delay_length = x->sr * x->maximum_delay_time + 1;
   x->delay_bytes = x->delay_length * sizeof(float);
   x->delay_line = (float *) getbytes(x->delay_bytes);
   if(x->delay_line == NULL){
         error("vdelay~: cannot allocate %d bytes of memory",
               x->delay_bytes);
         return NULL;
   }
   for(i = 0; i < x->delay_length; i++){
         x->delay_line[i] = 0.0;
   }
   x->feedback = feedback;
   x->write_index = 0;
   return x;
}
```

Figure 4.36 The Pd version of the new instance routine for *vdelay~*

```
void vdelay_dsp(t_vdelay *x, t_signal **sp)
{
   int i;
   int oldbytes = x->delay_bytes;
   x->delaytime_connected = 1;
   x->feedback_connected = 1;
   if(x->sr != sp[0]->s_sr){
         x->sr = sp[0]->s_sr;
         x->delay_length = x->sr * x->maximum_delay_time + 1;
         x->delay_bytes = x->delay_length * sizeof(float);
         x->delay_line = (float *) resizebytes(
               (void *)x->delay_line,oldbytes,x->delay_bytes);
         if(x->delay_line == NULL){
               error("vdelay~: cannot realloc %d bytes of memory",
                           x->delay_bytes);
               return;
         }
         for(i = 0; i < x->delay_length; i++){
               x->delay_line[i] = 0.0;
         }
         x->write_index = 0;
   }
   dsp_add(vdelay_perform, 6, x, sp[0]->s_vec, sp[1]->s_vec,
         sp[2]->s_vec, sp[3]->s_vec, sp[0]->s_n);
}
```

Figure 4.37 The Pd version of the dsp method for *vdelay~*

```
void vdelay_free(t_vdelay *x)
{
    freebytes(x->delay_line, x->delay_bytes);
}
```

Figure 4.38 The Pd version of the free function for *vdelay~*

```
#define TWOPI 6.28318530717959
int length = 1024;
int i;
float wavetable[LENGTH];
float phase;
main(){
for(i = 0; i < length; i++){
  phase = TWOPI * (float) i / (float) length;
  wavetable[i] = sin(phase);
 }
}
```

Figure 5.1 Generating and storing a digital sine wave

```
for( i = 0; i < length; i++ ){
  wavetable[i] = sin(TWOPI * (float) i / (float) length);
}
```

Figure 5.2 A more compact loop for generating a sine wave

```
twopi = 8 * atan(1);
```

Figure 5.4 Programmatically generating 2π

```
for(i = 0; i < length; i++){
  wavetable[i] = amplitudes[0];
}
```

Figure 5.5 Setting the DC component in code

```
phase = TWOPI * (float) i / (float) length;
```

Figure 5.8 Computing the running phase of a digital sine wave

```
phase = TWOPI * (float) j * (float) i / (float) length;
```

Figure 5.9 Computing the running phase for multiple harmonics

```
for(j = 1; j < harmonic_count; j++){
  for(i = 0; i < length; i++){
    phase = TWOPI * (float) j * (float) i / (float) length ;
    wavetable[i] += amplitudes[j] * sin(phase);
  }
}
```

Figure 5.10 Adding harmonics to the wavetable

```
for(j = 1; j < harmonic_count; j++){
  j2piolen = (float) j * TWOPI / (float) length;
  for(i = 0; i < length; i++){
    wavetable[i] += amplitudes[j] * sin(j2piolen * (float)i);
  }
}
```

Figure 5.11 A more efficient coding for adding harmonics to the wavetable

```
if(amplitudes[j] != 0.0)
```

Figure 5.12 A test for making sure the amplitude of a harmonic is not zero

```
if(amplitudes[j])
```

Figure 5.13 A more compact test condition

```
for(i = 0; i < length; i++){
  wavetable[i] = amplitudes[0];
}
for(j = 1; j < harmonic_count; j++){
  if(amplitudes[j]){
    j2piolen = (float) j * TWOPI / (float) length;
    for(i = 0; i < length; i++){
      wavetable[i] += amplitudes[j] * sin(j2piolen * (float)i);
    }
  }
}
```

Figure 5.14 Summing all components of the waveform

```
float max = 0.0;
for(i = 0; i < table_length; i++){
  if(max < fabs(wavetable[i]){
    max = fabs(wavetable[i]);
  }
}

if(max == 0.0) { //avoid divide by zero error
  /* could send an error message here */
  return;
}
rescale = 1.0 / max;
for(i = 0; i < length; i++){
  wavetable [i] *= rescale;
}
```

Figure 5.15 A normalizing algorithm

```
for(i = 0; i < table_length/2 ; i++){
  if(max < fabs(wavetable[i])){
    max = fabs(wavetable[i]);
  }
}
```

Figure 5.17 More efficient symmetry, assuming sine phase for all harmonics

```
if(max == 0.0){
  error("all zero wavetable!");
    return;
}
rescale = 1.0 / max ;
for( j = 0; j < table_length; j++ ){
  wavetable[j] *= rescale ;
}
```

Figure 5.18 Rescaling

```
sampling_increment =  frequency * table_length / sampling_rate
```

Figure 5.19 Calculating the sampling increment

```
#define OSCIL_DEFAULT_TABLESIZE 8192
#define OSCIL_DEFAULT_HARMS 10
#define OSCIL_MAX_HARMS 1024
#define OSCIL_DEFAULT_FREQUENCY 440.0
#define OSCIL_DEFAULT_WAVEFORM "sine"
#define OSCIL_MAX_TABLESIZE 1048576
```

Figure 5.20 Defining constants for *oscil~*

```
typedef struct _oscil
{
     t_pxobject obj; // required for all Max/MSP objects
     long table_length; // length of wavetable
     float *wavetable; // wavetable
     float *amplitudes; // list of amplitudes for each harmonic
     t_symbol *waveform; // the waveform used currently
     long harmonic_count; // number of harmonics
     float phase; // phase
     float si; // sampling increment
     float si_factor; // factor for generating sampling increment
     long bl_harms; // number of harmonics for band limited waveforms
     float piotwo; // pi over two
     float twopi; // two times pi
     float sr; // sampling rate
     long wavetable_bytes;// number of bytes stored in wavetable
     long amplitude_bytes;// number of bytes stored in amplitude
table
} t_oscil;
```

Figure 5.21 The object structure for *oscil~*

```
oscil_class = class_new("oscil~", (method)oscil_new,
            (method)dsp_free, sizeof(t_oscil), 0,A_GIMME,0);
```

Figure 5.23 The class instantiation call for *oscil~*

```
void *oscil_new(t_symbol *s, short argc, t_atom *argv);
```

Figure 5.24 The function prototype for the new instance routine

```
union word        /* union for packing any above datum */
{
  long w_long;
  float w_float;
  struct symbol *w_sym;
  struct object *w_obj;
};

typedef struct atom      /* and an atom which is a typed datum */
{
  short a_type;          /* from the above defs */
  union word a_w;
} t_atom, Atom;
```

Figure 5.25 Max/MSP type definitions for `word` and `atom`

```
void *oscil_new(t_symbol *s, short argc, t_atom *argv)
{
    float init_freq;
    t_oscil *x = (t_oscil *)object_alloc(oscil_class);
    dsp_setup((t_pxobject *)x,1);
    outlet_new((t_pxobject *)x, "signal");
    init_freq = 440.0;
    x->table_length = 8192;
    x->bl_harms = 10;
    x->waveform = gensym(OSCIL_DEFAULT_WAVEFORM);
    atom_arg_getfloat(&init_freq,0,argc,argv);
    atom_arg_getlong(&x->table_length,1,argc,argv);
    atom_arg_getsym(&x->waveform,2,argc,argv);
    atom_arg_getlong(&x->bl_harms,3,argc,argv);
    if(fabs(init_freq) > 1000000){
          init_freq = OSCIL_DEFAULT_FREQUENCY;
    }
    if(x->table_length < 4 || x->table_length > OSCIL_MAX_TABLESIZE){
          x->table_length = OSCIL_DEFAULT_TABLESIZE;
    }
    if(x->bl_harms < 0 || x->bl_harms > OSCIL_MAX_HARMS){
          x->bl_harms = OSCIL_DEFAULT_HARMS;
    }
    x->twopi = 8.0 * atan(1.0);
    x->piotwo = 2. * atan(1.0);
    x->wavetable_bytes = x->table_length * sizeof(float);
    x->wavetable = (float *) sysmem_newptr(x->wavetable_bytes);
    x->amplitude_bytes = OSCIL_MAX_HARMS*sizeof(float);
    x->amplitudes = (float *)sysmem_newptr(x->amplitude_bytes);
    x->phase = 0;
    x->sr = sys_getsr();
    x->si_factor = (float) x->table_length / x->sr;
    x->si = init_freq * x->si_factor;
    x->amplitudes[0] = 0.0;
    x->amplitudes[1] = 1.0;
    x->harmonic_count = 2;
    oscil_build_waveform(x);
    post("oscil~: freq:%f length:%d harms:%d waveform:%s",
    init_freq, x->table_length, x->bl_harms, x->waveform->s_name);
    return x;
}
```

Figure 5.26 The new instance routine for *oscil~*

```
void oscil_build_waveform(t_oscil *x) {
    float rescale;
    int i, j;
    float max;
    float *wavetable = x->wavetable;
    float *amplitudes = x->amplitudes;
    int partial_count = x->harmonic_count + 1;
    int table_length = x->table_length;
    float twopi = x->twopi;
    if(partial_count < 1){
        error("no harmonics specified, waveform not created.");
        return;
    }
    max = 0.0;
    for(i = 0; i < partial_count; i++){
        max += fabs(amplitudes[i]);
    }
    if(! max){
        error("all zero function, waveform not created.");
        return;
    }
    for(i = 0; i < table_length; i++){
        wavetable[i] = amplitudes[0];
    }
    for(i = 1 ; i < partial_count; i++){
        if(amplitudes[i]){
            for(j = 0; j < table_length; j++){
                wavetable[j] += amplitudes[i] * sin(twopi *
                ((float)i * ((float)j/(float)table_length)));
            }
        }
    }
    max = 0.0;
    for(i = 0; i < table_length / 2; i++){
        if(max < fabs(wavetable[i])){
            max = fabs(wavetable[i]) ;
        }
    }
    if(max == 0) {
        post("oscil~: weird all zero error - exiting!");
        return;
    }
    rescale = 1.0 / max;
    for(i = 0; i < table_length; i++){
        wavetable[i] *= rescale ;
    }
}
```

Figure 5.27 The waveform-building function for *oscil~*

```
t_int *oscil_perform(t_int *w)
{
   t_oscil *x = (t_oscil *) (w[1]);
   float *frequency = (t_float *)(w[2]);
   float *out = (t_float *)(w[3]);
   int n = w[4];
   float si_factor = x->si_factor;
   float si = x->si ;
   float phase = x->phase;
   int table_length = x->table_length;
   float *wavetable = x->wavetable;
   long iphase;
   while (n--) {
         si = *frequency++ * si_factor;
         iphase = trunc(phase);
         *out++ = wavetable[iphase];
         phase += si;
         while(phase >= table_length) {
               phase -= table_length;
         }
         while(phase < 0) {
               phase += table_length;
         }
   }
   x->phase = phase;
   return w + 5;
}
```

Figure 5.28 First version of the perform routine for *oscil~*

```
void oscil_assist(t_oscil *x, void *b, long msg, long arg,
   char *dst)
{
   if (msg == ASSIST_INLET) {
         sprintf(dst,"(signal/float) Frequency");
   }
   else if (msg == ASSIST_OUTLET) {
         sprintf(dst,"(signal) Output");
   }
}
```

Figure 5.30 The assist method

```
void oscil_float(t_oscil *x, double f) {
  x->si = f * x->si_factor;
}
```

Figure 5.31 The float method

```
t_int *oscil_perform2(t_int *w)
{
   t_oscil *x = (t_oscil *) (w[1]);
   float *out = (t_float *)(w[2]);
   int n = w[3];
   float si = x->si;
   float phase = x->phase;
   int table_length = x->table_length;
   float *wavetable = x->wavetable;
   long iphase;
   while (n--) {
         iphase = trunc(phase);
         *out++ = wavetable[iphase];
         phase += si;
         while(phase >= table_length) {
               phase -= table_length;
         }
         while(phase < 0) {
               phase += table_length;
         }
   }
   x->phase = phase;
   return w + 4;
}
```

Figure 5.32 The perform routine for when signal is not connected to the inlet

```
void oscil_dsp(t_oscil *x, t_signal **sp, short *count)
{
   if(count[0]) {
         dsp_add(oscil_perform, 4, x, sp[0]->s_vec,
         sp[1]->s_vec,sp[0]->s_n);
   }
   else {
         dsp_add(oscil_perform2, 3, x, sp[1]->s_vec, sp[0]->s_n);
   }
}
```

Figure 5.33 Selecting the appropriate perform routine

```
if(x->sr != sp[0]->s_sr){
    x->si *= x->sr / sp[0]->s_sr; // rescale sampling increment
    x->sr = sp[0]->s_sr; // assign new sampling rate
    x->si_factor = (float) x->table_length / x->sr;
}
```

Figure 5.34 Adjusting the sampling increment when the sampling rate changes

```
void oscil_sine(t_oscil *x)
{
  x->amplitudes[0] = 0.0;
  x->amplitudes[1] = 1.0;
  x->harmonic_count = 2;
  oscil_build_waveform(x);
}
```

Figure 5.35 The method to build a sine wave

```
void oscil_triangle(t_oscil *x)
{
  int i;
  float sign = 1.0;
  x-> amplitudes [0] = 0.0; // DC
  x->harmonic_count = x->bl_harms;
  for( i = 1 ; i < x->bl_harms; i += 2 ){
    x->amplitudes[i] = sign * 1.0/((float)i * (float)i);
    x->amplitudes[i + 1] = 0.0;
    sign *= -1;
  }
  oscil_build_waveform(x);
}
```

Figure 5.38 The triangle wave method

```
void oscil_sawtooth(t_oscil *x)
{
  int i;
  float sign = 1.0;

  x->amplitudes[0] = 0.0;
  x->harmonic_count = x->bl_harms;
  for(i = 1 ; i < x->bl_harms; i++){
    x->amplitudes[i] = sign * 1.0/(float)i;
    sign *= -1. ;
  }
  oscil_build_waveform(x);
}
```

Figure 5.40 The sawtooth wave method

```
void oscil_square(t_oscil *x)
{
  int i;
  x-> amplitudes [0] = 0.0;
  x->harmonic_count = x->bl_harms;
  for(i = 1 ; i < x->bl_harms; i += 2){
    x->amplitudes[i] = 1.0/(float)i;
    x->amplitudes[i + 1] = 0.0;
  }
  oscil_build_waveform(x);
}
```

Figure 5.42 The square wave method

```
void oscil_pulse(t_oscil *x)
{
  int i;
  x->amplitudes[0] = 0.0;
  x->harmonic_count = x->bl_harms;
  for(i = 1 ; i < x->bl_harms; i++){
    x->amplitudes[i] = 1.0;
  }
  oscil_build_waveform(x);
}
```

Figure 5.43 The pulse wave method

```
t_class *c;
oscil_class = class_new("oscil~", (method)oscil_new,
          (method)oscil_free, sizeof(t_oscil), 0,A_GIMME,0);
c = oscil_class;
class_addmethod(c,(method)oscil_sine, "sine", 0);
class_addmethod(c,(method)oscil_triangle, "triangle", 0);
class_addmethod(c,(method)oscil_square, "square", 0);
class_addmethod(c,(method)oscil_sawtooth, "sawtooth", 0);
class_addmethod(c,(method)oscil_pulse, "pulse", 0);
```

Figure 5.44 Waveform generation method bindings

```
void oscil_list (t_oscil *x, t_symbol *msg, short argc,
   t_atom *argv)
{
   short i;
   int harmonic_count = 0;
   float *amplitudes = x->amplitudes;
   for (i=0; i < argc; i++) {
         amplitudes[harmonic_count++] = atom_getfloat(argv + i);
   }
   x->harmonic_count = harmonic_count;
   oscil_build_waveform(x);
}
```

Figure 5.46 A list method to set the harmonic weightings

```
class_addmethod(c,(method)oscil_list, "list", A_GIMME, 0);
```

Figure 5.47 Binding the “list” message

```
if (x->waveform == gensym("sine")) {
          oscil_sine(x);
}
else if (x->waveform == gensym("triangle")) {
          oscil_triangle(x);
}
else if (x->waveform == gensym("square")) {
          oscil_square(x);
}
else if (x->waveform == gensym("sawtooth")) {
          oscil_sawtooth(x);
}
else if (x->waveform == gensym("pulse")) {
          oscil_pulse(x);
}
else {
          error("%s not a legal waveform - using sine wave instead",
                x->waveform->s_name);
          oscil_sine(x);
}
```

Figure 5.49 Selecting and generating the user-selected initial waveform

```
float *old_wavetable;
short dirty;
```

Figure 5.51 New object structure components to facilitate waveform transitions

```
x->dirty = 0;
x->old_wavetable = (float *) sysmem_newptr(x->wavetable_bytes);
```

Figure 5.52 Initializing components needed to transition between waveforms

```
void oscil_build_waveform(t_oscil *x) {
   // some variable declarations omitted here
   float *wavetable = x->wavetable;
   float *old_wavetable = x->old_wavetable;

   // copy current wave table to old wave table
   for(i = 0; i < table_length ; i++){
         old_wavetable[i] = wavetable[i];
   }
   x->dirty = 1;
   // new wave table generation omitted here
   x->dirty = 0;
}
```

Figure 5.53 Additional code for the wavetable building method

```
memcpy(old_wavetable, wavetable, table_length * sizeof(float));
```

Figure 5.54 A more compact way to copy blocks of memory

```
t_int *oscil_perform(t_int *w)
{
   // other variable declarations omitted
   float *old_wavetable = x->old_wavetable;
   // most of the existing code is omitted from this example
   while(n--)
         if(x->dirty){
               *out++ = old_wavetable[iphase];
         } else {
               *out++ = wavetable[iphase];
         }
   }
}
```

Figure 5.55 Revised code in the perform routines

```
crossfade_samples = crossfade_duration * srate / 1000.0;
crossfade_countdown = crossfade_samples;

while(crossfade_countdown--){
  fraction = crossfade_countdown/crossfade_samples;
  output = fraction * old_sample + (1 - fraction) * new_sample;
}
```

Figure 5.56 A sketch of the crossfade code

```
output = new_sample + fraction * (old_sample - new_sample);
```

Figure 5.57 A more efficient interpolation algorithm

```
int iphase;
// code omitted
iphase = phase;

if( crossfade_countdown ){
  fraction = (float)xfade_countdown/(float)xfade_samples;
  *out++ = wavetable[iphase] +
  fraction * (old_wavetable[iphase] - wavetable[iphase]);
 --xfade_countdown;
}
```

Figure 5.58 Implementing the crossfade in the perform routine

```
if( xfade_countdown ){
  fraction = PIOVERTWO *
    (float)xfade_countdown/(float)xfade_samples;
  *out++ = sin(fraction) * old_wavetable[iphase] + cos(fraction) *
          wavetable[iphase];
  --xfade_countdown;
}
```

Figure 5.59 A sketch of the equal-power crossfade code

```
#define OSCIL_NOFADE 0
#define OSCIL_LINEAR 1
#define OSCIL_POWER 2
```

Figure 5.61 Crossfade type constants

```
float xfade_duration;
int xfade_samples;
int xfade_countdown;
short xfadetype;
```

Figure 5.62 New crossfade components for the object structure

```
x->xfade_countdown = 0;
x->xfade_duration = 50. ;
x->xfade_samples = x->xfade_duration * x->sr / 1000.0;
x->xfadetype = OSCIL_LINEAR;
```

Figure 5.63 Initializations in the new instance routine

```
if(x->fadetype){
  x->fade_countdown = x->fade_samples;
}
```

Figure 5.64 Initiate a crossfade when the fade type is non-zero

```
short firsttime;
```

Figure 5.65 The initialization flag component

```
x->firsttime = 1;
```

Figure 5.66 Set the initialization flag in the new instance routine

```
x->firsttime = 0;
```

Figure 5.67 Permanently set the initialization flag to zero

```
if(x->xfadetype  && ! x->firsttime){
  x->xfade_countdown = x->xfade_samples;
}
```

Figure 5.68 Revised countdown reset code

```
t_int *oscil_perform(t_int *w)
{
    t_oscil *x = (t_oscil *) (w[1]);
    float *frequency = (t_float *)(w[2]);
    float *out = (t_float *)(w[3]);
```

```
    int n = w[4];
    float si_factor = x->si_factor;
    float si = x->si;
    float phase = x->phase;
    int table_length = x->table_length;
    float *wavetable = x->wavetable;
    float *old_wavetable = x->old_wavetable;
    int xfade_countdown = x->xfade_countdown;
    int xfade_samples = x->xfade_samples;
    short xfadetype = x->xfadetype;
    float piotwo = x->piotwo;
    long iphase;
    float fraction;
    while (n--) {
        si = *frequency++ * si_factor;
        iphase = trunc(phase);
        if(x->dirty){
            *out++ = old_wavetable[iphase];
        }
        else if (xfade_countdown > 0) {
            fraction =
            (float)xfade_countdown--/(float)xfade_samples;
            if(xfadetype == OSCIL_LINEAR){
                *out++ = wavetable[iphase] + fraction *
                (old_wavetable[iphase] - wavetable[iphase]);
            }
            else if (xfadetype == OSCIL_POWER) {
                fraction *= piotwo;
                *out++ = sin(fraction) * old_wavetable[iphase]
                + cos(fraction) * wavetable[iphase];
            }
        }
        else {
            *out++ = wavetable[iphase];
        }
        phase += si;
        while(phase >= table_length) {
            phase -= table_length;
        }
        while(phase < 0) {
            phase += table_length;
        }
    }
    x->xfade_countdown = xfade_countdown;
    x->phase = phase;
    return w + 5;
}
```

Figure 5.69 The perform routine with crossfades implemented

```
void oscil_fadetime (t_oscil *x, double fade_ms)
{
  if(fade_ms < 0.0 || fade_ms > 600000.0){
    error("%f is not a legal fade time", fade_ms);
    fade_ms = 50.;
  }
  x->xfade_duration = fade_ms;
  x->xfade_samples = x->xfade_duration * x->sr / 1000.0;
}
```

Figure 5.70 The fadetime method

```
class_addmethod(oscil_class,(method)oscil_fadetime, "fadetime",
      A_FLOAT, 0);
```

Figure 5.71 Binding the fadetime method in `main()`

```
void oscil_fadetype(t_oscil *x, long ftype)
{

  if(ftype < 0 || ftype > 2) {
    error("unknown type of fade, selecting no fade");
    ftype = 0;
  }
  x->xfadetype = (short)ftype;
}
```

Figure 5.72 The fadetype method

```
class_addmethod(c,(method)oscil_fadetype,"fadetype",A_LONG, 0);
```

Figure 5.73 Binding the fadetype method

```
void oscil_free(t_oscil *x)
{
  dsp_free((t_pxobject *) x)
  sysmem_freeptr(x->wavetable);
  sysmem_freeptr(x->old_wavetable);
  sysmem_freeptr(x->amplitudes);
}
```

Figure 5.77 The custom free memory routine

```
oscil_class = class_new("oscil~", (method)oscil_new,
             (method)oscil_free, sizeof(t_oscil), 0,A_GIMME,0);
```

Figure 5.78 Replacing `dsp_free()` with `oscil_free()` in the class definition

```
#include "m_pd.h"
#include "math.h"
```

Figure 5.79 Pd header files

```
typedef struct _oscil
{
    t_object obj; // required for all Pd objects
    t_float x_f; // internally convert floats to signals
    // the rest is identical to Max/MSP structure
} t_oscil;
```

Figure 5.80 The Pd object structure

```
void oscil_fadetime (t_oscil *x, t_floatarg fade_ms);
void oscil_fadetype(t_oscil *x, t_floatarg ftype);
```

Figure 5.81 Revised function prototypes with floating point arguments

```
void oscil_fadetype(t_oscil *x, t_floatarg ftype)
{
    if(ftype < 0 || ftype > 2) {
          error("unknown type of fade, selecting no fade");
          ftype = 0;
    }
    x->xfadetype = (short) ftype;
}
```

Figure 5.82 The revised `oscil_fadetype()` method

```
void oscil_tilde_setup (void)
{
   t_class * c;
   oscil_class = class_new(gensym("oscil~"),
         (t_newmethod) oscil_new, (t_method)oscil_free,
         sizeof(t_oscil), 0, A_GIMME, 0);
   CLASS_MAINSIGNALIN(oscil_class, t_oscil, x_f);
   c = oscil_class;
   class_addmethod(c,(t_method)oscil_dsp, gensym("dsp"),0);
   class_addmethod(c,(t_method)oscil_mute, gensym("mute"),
         A_FLOAT, 0);
   class_addmethod(c,(t_method)oscil_sine, gensym("sine"), 0);
   class_addmethod(c,(t_method)oscil_triangle,
         gensym("triangle"), 0);
   class_addmethod(c,(t_method)oscil_square, gensym("square"), 0);
   class_addmethod(c,(t_method)oscil_sawtooth,
         gensym("sawtooth"), 0);
   class_addmethod(c,(t_method)oscil_pulse, gensym("pulse"), 0);
   class_addmethod(c,(t_method)oscil_list, gensym("list"),
         A_GIMME, 0);
   class_addmethod(c,(t_method)oscil_fadetime, gensym("fadetime"),
         A_FLOAT, 0);
   class_addmethod(c,(t_method)oscil_fadetype, gensym("fadetype"),
         A_FLOAT, 0);
   post("oscil~ from \"Designing Audio Objects\" by Eric Lyon");
```

Figure 5.83 The class definition routine

```
t_oscil *x = (t_oscil *)object_alloc(oscil_class);
dsp_setup((t_pxobject *)x,1);
outlet_new((t_pxobject *)x, "signal");
```

Figure 5.84 Max/MSP instantiation code for the object and its inlets/outlets

```
t_oscil *x = (t_oscil *) pd_new(oscil_class);
outlet_new(&x->obj, gensym("signal"));
```

Figure 5.85 The Pd equivalent to the Max/MSP instantiation code

```
void oscil_dsp(t_oscil *x, t_signal **sp)
{

   if(x->sr != sp[0]->s_sr){
         x->si *= x->sr / sp[0]->s_sr;
         x->sr = sp[0]->s_sr;
         x->si_factor = (float) x->table_length / x->sr;
         x->xfade_samples = x->xfade_duration * x->sr / 1000.0;
   }
   dsp_add(oscil_perform, 4, x, sp[0]->s_vec, sp[1]->s_vec,
         sp[0]->s_n);
}
```

Figure 5.86 The Pd dsp method for *oscil~*

```
x->wavetable = (float *) getbytes(x->wavetable_bytes);
x->amplitudes = (float *)getbytes(x->amplitude_bytes);
x->old_wavetable = (float *) getbytes(x->wavetable_bytes);
```

Figure 5.87 Memory allocation calls for Pd

```
void oscil_free(t_oscil *x)
{
    freebytes(x->wavetable, x->wavetable_bytes);
    freebytes(x->old_wavetable, x->wavetable_bytes);
    freebytes(x->amplitudes, x->amplitude_bytes);
}
```

Figure 5.88 The free function for Pd

```
typedef struct _retroseq
{
  t_pxobject x_obj;
  float *sequence; // store sequence of frequency values
  int sequence_length; // length of sequence
  int duration_samples; // duration of a note in samples
  float note_duration_ms; // duration of a note in milliseconds
  int counter; // countdown the note in samples
  int position; // position in sequence
  float sr; // sampling rate
} t_retroseq;
```

Figure 6.1 The *retroseq~* object structure

```
void retroseq_dsp(t_retroseq *x, t_signal **sp, short *count)
{
  dsp_add(retroseq_perform, 3, x, sp[0]->s_vec, sp[0]->s_n);
}
```

Figure 6.2 The *retroseq~* dsp method

```
#define MAX_SEQUENCE 1024
```

Figure 6.3 Defining the maximum sequence length

```
void *retroseq_new(void)
{
    t_retroseq *x = (t_retroseq *)object_alloc(retroseq_class);
    dsp_setup((t_pxobject *)x,0);
    outlet_new((t_pxobject *)x, "signal");
    x->sr =  sys_getsr();
    if(!x->sr){
          x->sr = 44100.0;
    }
    x->sequence =
    (float *) sysmem_newptr(MAX_SEQUENCE * sizeof(float));
    if(x->sequence == NULL){
          post("retroseq: memory allocation fail");
          return NULL;
    }
    x->position = 0;
    x->note_duration_ms = 1000.0;
    x->duration_samples = x->note_duration_ms * x->sr / 1000.;
    x->counter = x->duration_samples;
    x->sequence_length = 3;
    x->sequence[0] = 440;
    x->sequence[1] = 550;
    x->sequence[2] = 660;
    return x;
}
```

Figure 6.4 The new instance routine for *retroseq~*

```
void retroseq_free(t_retroseq *x)
{
    dsp_free((t_pxobject *)x);
    sysmem_freeptr(x->sequence);
}
```

Figure 6.5 The free function for *retroseq~*

```
while(n--){
  counter--;
  *out++ = sequence[position];
}
```

Figure 6.6 Basic sequencing code

```
t_int *retroseq_perform(t_int *w)
{
    t_retroseq *x = (t_retroseq *) (w[1]);
    float *out = (t_float *)(w[2]);
    int n = w[3];
    int sequence_length = x->sequence_length;
    int duration_samples = x->duration_samples;
    int counter = x->counter;
    int position = x->position;
    float *sequence = x->sequence;

    while(n--){
          if(! counter--){
                ++position;
                if(position >= sequence_length){
                      position = 0;
                }
                counter = duration_samples;
          }
          *out++ = sequence[position];
    }
    x->counter = counter;
    x->position = position;
    return w + 4;
}
```

Figure 6.7 The perform routine for *retroseq~*

```
void *retroseq_new(void);
t_int *retroseq_perform(t_int *w);
void retroseq_dsp(t_retroseq *x, t_signal **sp, short *count);
```

Figure 6.8 Function prototypes for *retroseq~*

```
int main(void)
{
    t_class *c;
    retroseq_class = class_new("retroseq~", (method)retroseq_new,
          (method)retroseq_free, sizeof(t_retroseq), 0,A_GIMME,0);
    c = retroseq_class;
    class_addmethod(c, (method)retroseq_dsp, "dsp", A_CANT, 0);
    class_dspinit(c);
    class_register(CLASS_BOX, c);
    post("retroseq~: from \"Designing Audio Objects\" by Eric
Lyon");
    return 0;
}
```

Figure 6.9 The initialization routine

```
typedef struct _retroseq
{
  t_pxobject x_obj;
```

```
  float *sequence; // store sequence of frequency values
  long sequence_length; // length of sequence
  long duration_samples; // duration of a note in samples
  float note_duration_ms; // duration of a note in milliseconds
  long counter; // countdown the note in samples
  long position; // position in sequence
  float sr; // sampling rate
  float current_value; // stores current frequency (or whatever)
} t_retroseq;
```

Figure 6.11 The revised object structure for *retroseq~*

```
t_int *retroseq_perform(t_int *w)
{
    t_retroseq *x = (t_retroseq *) (w[1]);
    float *out = (t_float *)(w[2]);
    int n = w[3];
    long sequence_length = x->sequence_length;
    long duration_samples = x->duration_samples;
    long counter = x->counter;
    long position = x->position;
    float *sequence = x->sequence;
    float current_value = x->current_value;

    while(n--){
        if(! counter--){
            ++position;
            if(position >= sequence_length)
                position = 0;
            counter = duration_samples;
            current_value = sequence[position];
        }
        *out++ = current_value;
    }
    x->current_value = current_value;
    x->counter = counter;
    x->position = position;
    return w + 4;
}
```

Figure 6.12 The revised perform routine for *retroseq~*

```
void *retroseq_new(void)
{
    t_retroseq *x = (t_retroseq *)object_alloc(retroseq_class);
    dsp_setup((t_pxobject *)x,0);
    outlet_new((t_pxobject *)x, "signal");
    x->sr =  sys_getsr(); // will recheck in dsp method
    if(!x->sr){
          x->sr = 44100.0; // for safety
    }
    x->sequence = (float *) sysmem_newptr(MAX_SEQUENCE *
          sizeof(float));
    if(x->sequence == NULL){
          post("retroseq: memory allocation fail");
          return NULL;
    }
    x->position = 0 ;
    x->note_duration_ms = 250.0;
    x->duration_samples = x->note_duration_ms * x->sr / 1000. ;
    x->counter = x->duration_samples;
    x->sequence_length = 3;
    x->sequence[0] = 440;
    x->sequence[1] = 550;
    x->sequence[2] = 660;
    x->current_value = x->sequence[0];
    return x;
}
```

Figure 6.13 The revised new instance routine for *retroseq~*

```
void retroseq_list(t_retroseq *x, t_symbol *msg, short argc,
    t_atom *argv)
{
    int i;
    float *sequence = x->sequence;
    if( argc < 2 ){
          post("retroseq: sequence must have at least two
members");
          return;
    }
    x->sequence_length = argc;
    for (i=0; i < argc; i++) {
          sequence[i] = atom_getfloatarg(i,argc,argv);
    }
    x->position = x->sequence_length - 1;
}
```

Figure 6.14 The list processing method for reading sequences

```
class_addmethod(c,(method)retroseq_list,"list",A_GIMME,0);
```

Figure 6.15 Binding the list method

```
void retroseq_dsp(t_retroseq *x, t_signal **sp, short *count)
{
   if( x->sr != sp[0]->s_sr ){
         if(! sp[0]->s_sr){
               error("retroseq: zero sampling rate!");
               return;
         }
         x->counter *= x->sr / sp[0]->s_sr; // rescale countdown
         x->sr = sp[0]->s_sr; // assign new sampling rate
         x->duration_samples = x->note_duration_ms * 0.001 *
               x->sr;
   }
   dsp_add(retroseq_perform, 3, x, sp[0]->s_vec, sp[0]->s_n);
}
```

Figure 6.16 Adjusting to changes in the sampling rate inside the dsp method

```
x->note_duration_ms = 1000. * 60.0 / tempo;
```

Figure 6.18 Calculating the duration in milliseconds from the tempo

```
typedef struct _retroseq
{
  t_pxobject x_obj;
  float *sequence; // store sequence of frequency values
  int sequence_length; // length of sequence
  int duration_samples; // duration of a note in samples
  float note_duration_ms; // duration of a note in milliseconds
  int counter; // countdown the note in samples
  int position; // position in sequence
  float sr; // sampling rate
  float current_value; // stores current frequency (or whatever)
  float tempo; // tempo in BPM
} t_retroseq;
```

Figure 6.19 The revised object structure with new tempo-related components

```
void retroseq_tempo(t_retroseq *x, t_symbol *msg, short argc,
    t_atom *argv)
{
    float t;
    if(argc == 1){
          t = atom_getfloatarg(0, argc,argv);
    } else {
          return;
    }
    if( t <= 0 ){
          error("retroseq~: tempo must be greater than zero");
          return;
    }
    x->tempo = t;
    x->note_duration_ms = 0.25 * 60000.0 /  x->tempo ;
    x->duration_samples = x->note_duration_ms * x->sr / 1000.0;
}
```

Figure 6.20 The *retroseq~* tempo method

```
class_addmethod(c, (method)retroseq_tempo, "tempo", A_GIMME, 0);
```

Figure 6.21 Binding the tempo method

```
x->tempo = 60.0;
x->note_duration_ms = 60000.0 / x->tempo;
```

Figure 6.22 Initializing the tempo to 60 bpm

```
void retroseq_tempo(t_retroseq *x, t_symbol *msg, short argc,
    t_atom *argv)
{
    if(argc >= 1){
          t = atom_getfloatarg(0, argc,argv);
    }
    else {
          return;
    }
    if( t <= 0 ){
          error("retroseq~: tempo must be greater than zero");
          return;
    }
    x->tempo = t;
    x->note_duration_ms = 0.25 * 60000.0 /  x->tempo ;
    x->duration_samples = x->note_duration_ms * x->sr / 1000.0;
    if(x->counter > x->duration_samples){
          x->counter = x->duration_samples;
    }
}
```

Figure 6.24 Making tempo changes instantaneous

```
typedef struct _retroseq
{
  t_pxobject x_obj;
  float *f_sequence; // store sequence of frequency values
  float *d_sequence; // store sequence of duration values
  int f_sequence_length; // length of frequency sequence
  int d_sequence_length; // length of duration sequence
  int counter; // countdown the note in samples
  int f_position; // position in frequency sequence
  int d_position; // position in duration sequence
  float sr; // sampling rate
  float current_value; // stores current frequency (or whatever)
  int current_duration_samples; // current duration in samples
  float duration_factor; // get samples from duration, sr and tempo
  float tempo; // tempo in BPM
} t_retroseq;
```

Figure 6.26 The object structure modified to accept a duration sequence

```
void *retroseq_new(t_symbol *s, short argc, t_atom *argv)
{
   t_retroseq *x = (t_retroseq *)object_alloc(retroseq_class);
   dsp_setup((t_pxobject *)x,0);
   outlet_new((t_pxobject *)x, "signal");
   x->sr =  sys_getsr(); // will recheck in dsp method
   if(!x->sr){
         x->sr = 44100.0; // for safety
   }
   x->f_sequence =
         (float *)sysmem_newptr(MAX_SEQUENCE*sizeof(float));
   x->d_sequence =
         (float *)sysmem_newptr(MAX_SEQUENCE*sizeof(float) );
   if(x->f_sequence == NULL || x->d_sequence == NULL){
         post("retroseq: memory allocation fail");
         return NULL;
   }
   x->f_position = 0;
   x->d_position = 0;
   x->tempo = 60.0;
   x->duration_factor = x->sr/1000.0 ; // default tempo is 60
   x->f_sequence_length = 3;
   x->d_sequence_length = 3;
   x->f_sequence[0] = 440;
   x->f_sequence[1] = 550;
   x->f_sequence[2] = 660;
   x->d_sequence[0] = 250;
   x->d_sequence[1] = 125;
   x->d_sequence[2] = 125;
   x->current_value = x->f_sequence[0];
   x->counter = x->d_sequence[0] * x->sr/ 1000.0 ;
   return x;
}
```

Figure 6.27 The revised new instance routine for *retroseq~*

```
void retroseq_list(t_retroseq *x,t_symbol *msg,short argc,
    t_atom *argv)
{
    int i, j;
    float *f_sequence = x->f_sequence;
    float *d_sequence = x->d_sequence;

    if( argc % 2 ){ // reject lists with odd number of members
        error("retroseq~: odd number of arguments!");
        return;
    }
    if( argc < 2 )
        return;
    x->f_sequence_length = argc / 2;
    x->d_sequence_length = argc / 2;

    if(x->f_sequence_length >= MAX_SEQUENCE){
        error("retroseq~: sequence is too long");
        return;
    }
    for (i=0, j=0; i < argc; i += 2, j++) {
        atom_arg_getfloat(f_sequence+j, i, argc,argv);
        atom_arg_getfloat(d_sequence+j, i+1, argc,argv);
        if(d_sequence[j] <= 0){
            error("retroseq~: %f is an illegal duration value. Reset to 100 ms.",d_sequence[j] );
            d_sequence[j] = 100.0;
        }
    }
    x->f_position = x->f_sequence_length - 1;
    x->d_position = x->d_sequence_length - 1;
}
```

Figure 6.28 The revised list method incorporating a duration sequence

```
void retroseq_tempo(t_retroseq *x, t_symbol *msg, short argc,
    t_atom *argv)
{
    float old_tempo;
    float t;
    if(argc == 1){
        t = atom_getfloatarg(0, argc,argv);
    }
    else {
        return;
    }
    if( t <= 0 ){
        error("retroseq~: tempo must be greater than zero");
        return;
    }
    old_tempo = x->tempo;
    x->tempo = t;
    x->duration_factor = (60.0/x->tempo)*(x->sr/1000.0);
    x->counter *= old_tempo / x->tempo;
}
```

Figure 6.29 Making tempo changes instantaneous in the tempo method

```
void retroseq_dsp(t_retroseq *x, t_signal **sp, short *count)
{
    if( x->sr != sp[0]->s_sr ){
          if( ! sp[0]->s_sr ){
                error("zero sampling rate!");
                return;
          }
          x->counter *= x->sr/sp[0]->s_sr; // rescale countdown
          x->duration_factor *=  sp[0]->s_sr/x->sr; // rescale factor
          x->sr = sp[0]->s_sr; // assign new sampling rate
    }
    dsp_add(retroseq_perform, 3, x, sp[0]->s_vec, sp[0]->s_n);
}
```

Figure 6.30 Making tempo changes instantaneous in the dsp method

```
t_int *retroseq_perform(t_int *w)
{
    t_retroseq *x = (t_retroseq *) (w[1]);
    float *out = (t_float *)(w[2]);
    int n = w[3];
    int f_sequence_length = x->f_sequence_length;
    int d_sequence_length = x->d_sequence_length;
    int counter = x->counter;
    int f_position = x->f_position;
    int d_position = x->d_position;
    float *f_sequence = x->f_sequence;
    float *d_sequence = x->d_sequence;
    float current_value = x->current_value;
    float duration_factor = x->duration_factor;

    while(n--){
          if(! counter--){
                if(++f_position >= f_sequence_length){
                      f_position = 0;
                }
                if(++d_position >= d_sequence_length){
                      d_position = 0;
                }
                counter = d_sequence[d_position] * duration_factor;
                current_value = f_sequence[f_position];

          }
          *out++ = current_value;
    }
    x->current_value = current_value;
    x->counter = counter;
    x->f_position = f_position;
    x->d_position = d_position;
    return w + 4;
}
```

Figure 6.31 The revised perform method

```
void retroseq_free(t_retroseq *x)
{
    dsp_free((t_pxobject *)x);
    sysmem_freeptr(x->d_sequence);
    sysmem_freeptr(x->f_sequence);
}
```

Figure 6.32 Updating the free function

```
void retroseq_freqlist(t_retroseq *x, t_symbol *msg, short argc,
    t_atom *argv)
{
    int i;
    float *f_sequence = x->f_sequence;
    if( argc < 2 ){
          return;
    }
    x->f_sequence_length = argc;
    if( x->f_sequence_length  >= MAX_SEQUENCE ){
          error("retroseq~: frequency sequence is too long");
          return;
    }
    for (i=0 ; i < argc; i++) {
          f_sequence[i] = atom_getfloatarg(i,argc,argv);
    }
    x->f_position = x->f_sequence_length - 1;
}
```

Figure 6.33 The data entry method for the frequency sequence

```
class_addmethod(c,(method)retroseq_durlist,"durlist", A_GIMME,0);
class_addmethod(c,(method)retroseq_freqlist,"freqlist", A_GIMME,0);
```

Figure 6.34 Binding the data entry methods for the duration and frequency sequences

```
void *list_outlet;
```

Figure 6.36 The list outlet component

```
x->list_outlet = listout((t_pxobject *)x);
dsp_setup((t_pxobject *)x,1);
outlet_new((t_pxobject *)x, "signal");
```

Figure 6.37 Initializing the outlets in the new instance routine

```
void *the_clock; // clock for non-signal events
```

Figure 6.38 The clock component of the object structure

```
x->the_clock = clock_new(x,(method)retroseq_send_adsr);
```

Figure 6.39 Initializing the clock in the new instance routine

```
t_atom *adsr_list;
```

Figure 6.43 The ADSR list component

```
x->adsr_list = (t_atom *) sysmem_getptr(10 * sizeof(t_atom));
```

Figure 6.44 Allocating memory for the ADSR list

```
float *adsr;
```

Figure 6.45 The ADSR data component for the object structure

```
float sustain_amplitude;
```

Figure 6.46 The sustain amplitude level component

```
x->adsr = (float *) sysmem_newptr (4 * sizeof(float));
```

Figure 6.47 Allocating memory for the ADSR data

```
x->sustain_amplitude = 0.7;
```

Figure 6.48 Initializing the amplitude sustain level

```
float *adsr_out;
```

Figure 6.49 The ADSR output data component

```
x->adsr_out = (float *) sysmem_newptr(10 * sizeof(float));
```

Figure 6.50 Allocating memory for the ADSR output data

```
x->adsr_out[0] = 0.0;
x->adsr_out[1] = 0.0;
x->adsr_out[2] = 1.0;
x->adsr_out[8] = 0.0;
```

Figure 6.51 Initializing the fixed amplitude data points

```
x->adsr[0] = 20;
x->adsr[1] = 50;
x->adsr[2] = 100;
x->adsr[3] = 50;
```

Figure 6.52 Initializing duration values of the ADSR

```
void retroseq_send_adsr(t_retroseq *x)
{
  t_atom *adsr_list = x->adsr_list;
  float *adsr = x->adsr;
  float *adsr_out = x->adsr_out;
  float note_duration_ms = x->note_duration_ms;
  int i;

  /* envelope data massaging omitted for now */

  for( i = 0; i < 10; i++ ){
    SETFLOAT(adsr_list+i,adsr_out[i]);
  }
  outlet_list(x->list_outlet,NULL,10,adsr_list);
}
```

Figure 6.53 Sending the ADSR data to the list outlet

```
void *outlet_list(void *o, t_symbol *s, short ac, t_atom *av);
```

Figure 6.54 The function prototype for `outlet_list()`

```
#define SETFLOAT(ap, x) ((ap)->a_type = A_FLOAT,(ap)->a_w.w_float = (x))
```

Figure 6.55 The Max/MSP `SETFLOAT()` macro (courtesy of Cycling ’74)

```
int x;
int *px;

x = 5;
px = &x;

/* pointer px now contains the address of x, so *px is equal to 5 */
```

Figure 6.56 Assigning a pointer address

```
int x[4];
int *px;

x[2] = 5;
px = &x[2];

/* now *px is 5 again */
```

Figure 6.57 Assigning an address within an array to a pointer

```
for(i = 0; i < 10; i++){
SETFLOAT(adsr_list+i,adsr_out[i]);
}
```

Figure 6.58 Using the `SETFLOAT()` macro

```
short elastic_sustain;
```

Figure 6.59 The flag for envelope time-scaling

```
x->elastic_sustain = 0;
```

Figure 6.60 Initializing the envelope time-scaling flag

```
void retroseq_send_adsr(t_retroseq *x)
{
  t_atom *adsr_list = x->adsr_list;
  float *adsr = x->adsr;
  float *adsr_out = x->adsr_out;
  float note_duration_ms;
  float duration_sum;
  short elastic_sustain = x->elastic_sustain;
  int d_position = x->d_position;
  float *d_sequence = x->d_sequence;
  float tempo = x->tempo;
  float rescale ;
  int i;

  note_duration_ms = d_sequence[d_position] * (60.0/tempo);
  adsr_out[4] = adsr_out[6] = x->sustain_amplitude;
  adsr_out[3] = adsr[0]; // attack duration
  adsr_out[5] = adsr[1]; // decay duration
  adsr_out[9] = adsr[3]; // release duration
  if(elastic_sustain){
    adsr_out[7] = note_duration_ms - (adsr[0]+adsr[1]+adsr[3]);
    if(adsr_out[7] < 1.0) // minimum sustain of 1 millisecond
      adsr_out[7] = 1.0 ;
  } else {
    adsr_out[7] = adsr[2]; // user specified sustain duration
  }
  duration_sum = adsr_out[3] + adsr_out[5] + adsr_out[7] +
         adsr_out[9];
  if(duration_sum > note_duration_ms){
    rescale = note_duration_ms / duration_sum ;
    adsr_out[3] *= rescale ;
    adsr_out[5] *= rescale ;
    adsr_out[7] *= rescale ;
    adsr_out[9] *= rescale ;
  }
  for(i = 0; i < 10; i++){
    SETFLOAT(adsr_list+i,adsr_out[i]);
  }
  outlet_list(x->list_outlet,NULL,10,adsr_list);
}
```

Figure 6.61 The `retroseq_send_adsr()` method

```
while(n--){
    if(! counter--){
        if(++f_position >= f_sequence_length){
            f_position = 0;
        }
        if(++d_position >= d_sequence_length){
            d_position = 0;
        }
        counter = d_sequence[d_position] * duration_factor;
        current_value = f_sequence[f_position];
        clock_delay(x->the_clock,0); // defer list output
    }
    *out++ = current_value;
}
```

Figure 6.62 Sending the ADSR data to an outlet from the DSP loop with `clock_delay()`

```
void retroseq_adsr(t_retroseq *x, t_symbol *msg, short argc, t_atom *argv)
{
    float *adsr = x->adsr;
    int i;

    if( argc < 4 ){
        error("not enough parameters for adsr (should be 4)");
        return;
    }
    for (i=0 ; i < 4; i++) {
        adsr[i] = atom_getfloatarg(i,argc,argv);
        if( adsr[i] < 1.0 ){
            adsr[i] = 1.0;
        }
    }
}
```

Figure 6.63 The input method for ADSR data

```
void retroseq_sustain_amplitude(t_retroseq *x, t_symbol *msg, short argc, t_atom *argv)
{
    if(argc >= 1){
        x->sustain_amplitude = atom_getfloatarg(0,argc,argv);
    }
}
```

Figure 6.64 Setting the sustain amplitude

```
void retroseq_elastic_sustain(t_retroseq *x, t_symbol *msg, short argc, t_atom *argv)
{
    if(argc >= 1){
          x->elastic_sustain = (short) atom_getfloatarg(0,argc,argv);
    }
}
```

Figure 6.65 Setting the sustain duration

```
class_addmethod(c, (method)retroseq_adsr, "adsr", A_GIMME, 0);
class_addmethod(c, (method)retroseq_sustain_amplitude,"sustain_amplitude", A_GIMME, 0);
class_addmethod(c, (method)retroseq_elastic_sustain,"elastic_sustain", A_GIMME, 0);
```

Figure 6.66 Binding the new ADSR-related methods

```
void retroseq_free(t_retroseq *x)
{
   dsp_free((t_pxobject *)x);
   sysmem_freeptr(x->d_sequence);
   sysmem_freeptr(x->f_sequence);
   sysmem_freeptr(x->adsr);
   sysmem_freeptr(x->adsr_out);
   sysmem_freeptr(x->adsr_list);
   object_free(x->the_clock);
}
```

Figure 6.67 The revised `retroseq_free()` function

```
void *bang_outlet; // start-of-sequence bang outlet
void *bang_clock; // clock for the bang
```

Figure 6.69 New components to send a bang from *retroseq~*

```
x->bang_outlet = bangout((t_pxobject *)x);
x->list_outlet = listout((t_pxobject *)x);
dsp_setup((t_pxobject *)x,0);
outlet_new((t_pxobject *)x, "signal");
```

Figure 6.70 Instantiating the outlets in the new instance routine

```
void retroseq_send_bang(t_retroseq *x)
{
  outlet_bang(x->bang_outlet);
}
```

Figure 6.71 The `retroseq_send_bang()` routine

```
x->bang_clock = clock_new(x,(method)retroseq_send_bang);
```

Figure 6.72 Binding the outlet bang routine

```
while(n--){
  if(! counter--){
    if(++f_position >= f_sequence_length){
      f_position = 0;
      clock_delay(x->bang_clock,0); // send a bang
    }
    if(++d_position >= d_sequence_length)
      d_position = 0;
    counter = d_sequence[d_position] * duration_factor ;
    current_value = f_sequence[f_position];
    clock_delay(x->the_clock,0); // defer list output
  }
  *out++ = current_value;
}
```

Figure 6.73 The perform loop revised to send a bang

```
object_free(x->list_clock);
object_free(x->bang_clock);
```

Figure 6.74 Freeing the clocks when the object is destroyed

```
void retroseq_assist (t_retroseq *x, void *b, long msg, long arg,
    char *dst)
{
    if (msg == ASSIST_INLET) {
          sprintf(dst,"(messages) ");

    } else if (msg == ASSIST_OUTLET) {
          switch (arg){
                case 0:
                      sprintf(dst,"(signal) Output");
                      break;
                case 1:
                      sprintf(dst,"(list) ADSR envelope");
                      break;
                case 2:
                      sprintf(dst,"(bang) Sequence Start Bang");
                      break;
          }
    }
}
```

Figure 6.75 The assist method for *retroseq~*

```
srand(clock());
```

Figure 6.78 Seeding the random number generator

```
int rpos;
float rand_member;

rpos = rand() % len;
rand_member = seq[rpos];
```

Figure 6.79 Extracting a random member of a sequence

```
void retroseq_permute(float *sequence, float *permutation,
    int len)
{
  int cnt = 0;
  int rpos;
  float tmp;
  int i;
  int tlen = len;

  while(tlen > 1){
    rpos = rand() % tlen ;
    permutation[cnt++] = sequence[rpos];
    tmp = sequence[rpos]; // swap here
    sequence[rpos] = sequence[tlen - 1];
    sequence[tlen - 1] = tmp;
    --tlen;
  }
  permutation[len - 1] = sequence[0];
  for( i = 0; i < len; i++ ){
    sequence[i] = permutation[i];
  }
}
```

Figure 6.80 The `retroseq_permute()` method

```
float *tmp_permutation;
```

Figure 6.81 A temporary work space component for the object structure

```
x->tmp_permutation = (float *)sysmem_newptr(MAX_SEQUENCE *
sizeof(float));
```

Figure 6.82 Allocating memory for the permutation work space

```
sysmem_freeptr(x->tmp_permutation);
```

Figure 6.83 Freeing memory

```
void retroseq_shuffle_freqs(t_retroseq *x)
{
  float *tmp_permutation = x->tmp_permutation;
  float *f_sequence = x->f_sequence;
  int f_sequence_length = x->f_sequence_length;
  retroseq_permute(f_sequence, tmp_permutation, f_sequence_length);
}
```

Figure 6.84 The `retroseq_shuffle_freqs()` method

```
void retroseq_shuffle_durs(t_retroseq *x)
{
  float *tmp_permutation = x->tmp_permutation;
  float *d_sequence = x->d_sequence;
  int d_sequence_length = x->d_sequence_length;
  retroseq_permute(d_sequence, tmp_permutation, d_sequence_length);
}
```

Figure 6.85 the `retroseq_shuffle_durs()` method

```
void retroseq_shuffle(t_retroseq *x)
{
  retroseq_shuffle_freqs(x);
  retroseq_shuffle_durs(x);
}
```

Figure 6.86 Combining both shuffle methods into a single method

```
class_addmethod(c, (method)retroseq_shuffle_freqs,
    "shuffle_freqs",0);
class_addmethod(c, (method)retroseq_shuffle_durs,
    "shuffle_durs",0);
class_addmethod(c, (method)retroseq_shuffle, "shuffle",0);
```

Figure 6.87 Binding the shuffle methods

```
void *f_plist_outlet; // outlet for permuted frequencies
void *d_plist_outlet; // outlet for permuted durations
t_atom *pseq_list; // holds permuted lists
```

Figure 6.88 Object components for new list outlets

```
x->pseq_list = (t_atom *) sysmem_newptr(MAX_SEQUENCE *
      sizeof(t_atom));
```

Figure 6.89 Allocating memory for the atom list

```
sysmem_freeptr(x->pseq_list);
```

Figure 6.90 Freeing memory for the atom list

```
x->d_plist_outlet = listout((t_pxobject *)x);
x->f_plist_outlet = listout((t_pxobject *)x);
x->bang_outlet = bangout((t_pxobject *)x);
x->list_outlet = listout((t_pxobject *)x);
dsp_setup((t_pxobject *)x,0);
outlet_new((t_pxobject *)x, "signal");
```

Figure 6.91 Revised outlet instantiation calls in the new instance routine

```
void retroseq_shuffle_freqs(t_retroseq *x)
{
  float *tmp_permutation = x->tmp_permutation;
  float *f_sequence = x->f_sequence;
  int f_sequence_length = x->f_sequence_length;
  t_atom *pseq_list = x->pseq_list;
  int i;

  retroseq_permute(f_sequence, tmp_permutation,
f_sequence_length);
  for( i = 0; i < f_sequence_length; i++ ){
         SETFLOAT(pseq_list+i,f_sequence[i]);
  }
  outlet_list(x->f_plist_outlet,NULL,f_sequence_length,pseq_list);
}
```

Figure 6.92 The updated frequency sequence shuffle method

```
void retroseq_shuffle_durs(t_retroseq *x)
{
  float *tmp_permutation = x->tmp_permutation;
  float *d_sequence = x->d_sequence;
  int d_sequence_length = x->d_sequence_length;
  t_atom *pseq_list = x->pseq_list;
  int i;

  retroseq_permute(d_sequence, tmp_permutation,
   d_sequence_length);
  for( i = 0; i < d_sequence_length; i++ ){
    SETFLOAT(pseq_list+i,d_sequence[i]);
  }
  outlet_list(x->d_plist_outlet,NULL,d_sequence_length,pseq_list);
}
```

Figure 6.93 The updated duration sequence shuffle method

```
void retroseq_assist (t_retroseq *x, void *b, long msg, long arg,
   char *dst)
{
   if (msg == ASSIST_INLET) {
         sprintf(dst,"(messages) ");

   } else if (msg == ASSIST_OUTLET) {
         switch (arg){
               case 0:
                     sprintf(dst,"(signal) Output");
                     break;
               case 1:
                     sprintf(dst,"(list) ADSR envelope");
                     break;
               case 2:
                     sprintf(dst,"(bang) Sequence Start Bang");
                     break;
               case 3:
                     sprintf(dst,"(list) Permuted Frequency
List");
                     break;
               case 4:
                     sprintf(dst,"(list) Permuted Duration List");
                     break;
         }
   }
}
```

Figure 6.94 The updated assist method

```
short manual_override; // toggle manual override
short trigger_sent; // user sent a bang
```

Figure 6.96 Object structure components for manual override

```
t_int *retroseq_perform(t_int *w)
{
    t_retroseq *x = (t_retroseq *) (w[1]);
    float *out = (t_float *)(w[2]);
    int n = w[3];
    int f_sequence_length = x->f_sequence_length;
    int d_sequence_length = x->d_sequence_length;
    int counter = x->counter;
    int f_position = x->f_position;
    int d_position = x->d_position;
    float *f_sequence = x->f_sequence;
    float *d_sequence = x->d_sequence;
    float current_value = x->current_value;
    float duration_factor = x->duration_factor;
    short manual_override = x->manual_override;
    short trigger_sent = x->trigger_sent;

    if( manual_override ){
        while(n--){
            if( trigger_sent ){
                trigger_sent = 0;
                ++f_position;
                if( f_position >= f_sequence_length ){
                    f_position = 0;
                    clock_delay(x->bang_clock,0);
                }
                current_value = f_sequence[f_position];
                clock_delay(x->list_clock,0);
            }
            *out++ = current_value;
        }
    }
    else {
        while(n--){
            if(! counter--){
                if(++f_position >= f_sequence_length){
                    f_position = 0;
                    clock_delay(x->bang_clock,0);
                }
                if(++d_position >= d_sequence_length){
                    d_position = 0;
                }
                counter = d_sequence[d_position] *
                    duration_factor ;
                current_value = f_sequence[f_position];
                clock_delay(x->list_clock,0);
            }
            *out++ = current_value;
        }
    }
    x->trigger_sent = trigger_sent;
    x->current_value = current_value;
    x->counter = counter;
    x->f_position = f_position;
    x->d_position = d_position;
    return w + 4;
}
```

Figure 6.97 The revised perform routine with manual override logic

```
void retroseq_manual_override(t_retroseq *x, long state)
{
  x->manual_override = (short) state;
}
```

Figure 6.98 The manual override method

```
void retroseq_bang(t_retroseq *x)
{
  x->trigger_sent = 1;
}
```

Figure 6.99 Using the Max *bang* message to send a manual trigger

```
class_addmethod(c, (method)retroseq_bang, "bang",0);
class_addmethod(c, (method)retroseq_manual_override,
    "manual_override", A_LONG, 0);
```

Figure 6.100 Binding the “bang” and “manual_override” methods

```
if(x->manual_override){
  adsr_out[7] = adsr[2]; // swap in user sustain
}
else {
  if(elastic_sustain){
    adsr_out[7] = note_duration_ms - (adsr[0]+adsr[1]+adsr[3]);
    if(adsr_out[7] < 1.0) // minimum sustain of 1 millisecond
      adsr_out[7] = 1.0 ;
  } else {
    adsr_out[7] = adsr[2]; // user specified sustain duration
  }
  duration_sum = adsr_out[3] + adsr_out[5] + adsr_out[7] + adsr_out[9];

  /* if note is shorter than total then rescale envelope */

  if(duration_sum > note_duration_ms){
    rescale = note_duration_ms / duration_sum ;
    adsr_out[3] *= rescale;
    adsr_out[5] *= rescale;
    adsr_out[7] *= rescale;
    adsr_out[9] *= rescale;
  }
}
```

Figure 6.102 Modifications to `retroseq_send_adsr()`

```
void retroseq_tilde_setup(void)
{
    t_class *c;
    retroseq_class = class_new(gensym("retroseq~"),
            (t_newmethod)retroseq_new,
            (t_method)retroseq_free, sizeof(t_retroseq),
                  0,A_GIMME,0);
    c = retroseq_class;
    class_addmethod(c, (t_method)retroseq_dsp, gensym("dsp"),0,
            A_CANT, 0);
    class_addmethod(c, (t_method)retroseq_list, gensym("list"),
            A_GIMME, 0);
    class_addmethod(c, (t_method)retroseq_durlist,
                  gensym("durlist"),
            A_GIMME, 0);
    class_addmethod(c, (t_method)retroseq_freqlist,
            gensym("freqlist"), A_GIMME, 0);
    class_addmethod(c, (t_method)retroseq_tempo, gensym("tempo"),
            A_GIMME, 0);
    class_addmethod(c, (t_method)retroseq_adsr, gensym("adsr"),
            A_GIMME, 0);
    class_addmethod(c, (t_method)retroseq_sustain_amplitude,
            gensym("sustain_amplitude"), A_GIMME, 0);
    class_addmethod(c, (t_method)retroseq_elastic_sustain,gensym(
            "elastic_sustain"), A_GIMME, 0);
    class_addmethod(c, (t_method)retroseq_shuffle_freqs,
            gensym("shuffle_freqs"),0);
    class_addmethod(c, (t_method)retroseq_shuffle_durs,
            gensym("shuffle_durs"),0);
    class_addmethod(c, (t_method)retroseq_shuffle,
            gensym("shuffle"),0);
    class_addmethod(c, (t_method)retroseq_manual_override,
            gensym("manual_override"), A_GIMME, 0);
    class_addmethod(c, (t_method)retroseq_bang, gensym("bang"),0);
    post("retroseq~ from \"Designing Audio Objects\" by Eric
Lyon");
}
```

Figure 6.104 The initialization routine

```
void retroseq_free(t_retroseq *x)
{
   freebytes(x->d_sequence, MAX_SEQUENCE*sizeof(float));
   freebytes(x->f_sequence, MAX_SEQUENCE*sizeof(float));
   freebytes(x->adsr, 4 * sizeof(float));
   freebytes(x->adsr_out, 10 * sizeof(float));
   freebytes(x->adsr_list, 10 * sizeof(t_atom));
   freebytes(x->tmp_permutation, MAX_SEQUENCE * sizeof(float));
   freebytes(x->pseq_list, MAX_SEQUENCE * sizeof(t_atom));
   clock_free(x->list_clock);
   clock_free(x->bang_clock);
}
```

Figure 6.105 Using Pd function calls in the `retroseq_free()` routine

```
void *retroseq_new(t_symbol *s, short argc, t_atom *argv)
{
   t_retroseq *x = (t_retroseq *)pd_new(retroseq_class);
   outlet_new(&x->obj, gensym("signal"));
   x->list_outlet = outlet_new(&x->obj, gensym("list"));
   x->bang_outlet = outlet_new(&x->obj, gensym("bang"));
   x->f_plist_outlet = outlet_new(&x->obj, gensym("list"));
   x->d_plist_outlet = outlet_new(&x->obj, gensym("list"));
   x->sr =  sys_getsr();
   if(!x->sr){
         x->sr = 44100.0;
   }
   x->list_clock = clock_new(x,(t_method)retroseq_send_adsr);
   x->bang_clock = clock_new(x,(t_method)retroseq_send_bang);
   x->f_sequence = (float *)getbytes(MAX_SEQUENCE*sizeof(float));
   x->d_sequence = (float *)getbytes(MAX_SEQUENCE*sizeof(float) );
   x->adsr_list = (t_atom *) getbytes(10 * sizeof(t_atom));
   x->adsr_out = (float *) getbytes(10 * sizeof(float));
   x->adsr = (float *) getbytes(4 * sizeof(float));
   x->tmp_permutation = (float *)getbytes(MAX_SEQUENCE *
         sizeof(float));
   x->pseq_list = (t_atom *) getbytes(MAX_SEQUENCE *
         sizeof(t_atom));
   if(x->f_sequence == NULL || x->d_sequence == NULL){
         post("retroseq~: memory allocation fail");
         return NULL;
   }
   srand(clock());
   x->f_position = 0;
   x->d_position = 0;
   x->elastic_sustain = 0;
   x->tempo = 60.0;
   x->duration_factor = x->sr/1000.0;
   x->f_sequence_length = 3;
   x->d_sequence_length = 3;
   x->f_sequence[0] = 440;
   x->f_sequence[1] = 550;
   x->f_sequence[2] = 660;
   x->d_sequence[0] = 250;
   x->d_sequence[1] = 125;
   x->d_sequence[2] = 125;
   x->adsr_out[0] = 0.0;
   x->adsr_out[1] = 0.0;
   x->adsr_out[2] = 1.0;
   x->adsr_out[8] = 0.0;
   x->adsr[0] = 20;
   x->adsr[1] = 50;
   x->adsr[2] = 100;
   x->adsr[3] = 50;
   x->sustain_amplitude = 0.7;
   x->current_value = x->f_sequence[0];
   x->counter = x->d_sequence[0] * x->sr/ 1000.0 ;
   return x;
}
```

Figure 6.106 The revised new instance routing for the Pd version of *retroseq~*

```
#include "ext.h"
#include "ext_obex.h"
#include "buffer.h"
```

Figure 7.1 Required header files for Max/MSP buffer operations

```
typedef struct _bed
{
      t_object    obj;
      t_symbol    *b_name;
      t_buffer    *buffy;
} t_bed;
```

Figure 7.2 The *bed* object structure

```
typedef struct symbol
{
      char *s_name;                 ///< name: a c-string
      struct object *s_thing; ///< possible binding to a t_object
} Symbol, t_symbol;
```

Figure 7.3 The Max/MSP symbol structure

```
int attach_buffer(t_bed *x)
{
   t_object *o;
   o = x->b_name->s_thing;
   if(o == NULL){
         object_post((t_object *)x,    "\"%s\" is not a valid
         buffer", x->b_name->s_name);
         return 0;
   }
   if (ob_sym(o) == gensym("buffer~")) {
         x->buffy = (t_buffer *) o;
         return 1;
   } else {
         return 0;
   }
}
```

Figure 7.4 Using a buffer symbol to gain access to the corresponding buffer object

```
void bed_info(t_bed *x)
{
   t_buffer *b;
   if( ! attach_buffer(x) ){
         post("bed: %s is not a valid buffer",x->b_name->s_name);
         return;
   }
   b = x->buffy;
   post("my name is: %s", b->b_name->s_name);
   post("my frame count is: %d", b->b_frames);
   post("my channel count is: %d", b->b_nchans);
   post("my validity is: %d", b->b_valid);
   post("my in use status is: %d", b->b_inuse);
}
```

Figure 7.5 Printing out buffer information

```
void bed_bufname(t_bed *x, t_symbol *name)
{     x->b_name = name; }
```

Figure 7.6 The bufname method

```
void *bed_new(t_symbol *s, short argc, t_atom *argv)
{
    t_bed *x = (t_bed *)object_alloc(bed_class);
    atom_arg_getsym(&x->b_name, 0, argc, argv);
    return x;
}
```

Figure 7.7 The new instance routine for *bed*

```
void bed_normalize(t_bed *x)
{
   t_buffer *b;
   float maxamp = 0.0;
   float rescale;
   int i;

   if(! attach_buffer(x)){
         object_post((t_object *)x,
         "normalize: %s is not a valid buffer",x->b_name->s_name);
         return;
   }
   b = x->buffy;
   for(i = 0; i < b->b_frames * b->b_nchans; i++){
         if(maxamp < fabs(b->b_samples[i]) ){
               maxamp = fabs(b->b_samples[i]);
         }
   }
   if(maxamp > 0.0000001){
         rescale = 1.0 / maxamp;
   }
   else {
   post("rescale: amplitude is too low to rescale: %f", maxamp);
         return;
   }
   for(i = 0; i < b->b_frames * b->b_nchans; i++){
         b->b_samples[i] *= rescale;
   }
}
```

Figure 7.9 The normalization method

```
void bed_normalize(t_bed *x)
{
   t_buffer *b;
   float maxamp = 0.0;
   float rescale;
   int i;

   if( ! attach_buffer(x) ){
         post("bed: %s is not a valid buffer",x->b_name->s_name);
         return;
   }
   b = x->buffy;
   ATOMIC_INCREMENT(&b->b_inuse);
   if (!b->b_valid) {
         ATOMIC_DECREMENT(&b->b_inuse);
         object_post((t_object *)x,
               "bed normalize: not a valid buffer!");
         return;
   }
   for(i = 0; i < b->b_frames * b->b_nchans; i++){
         if(maxamp < fabs(b->b_samples[i]) ){
               maxamp = fabs(b->b_samples[i]);
         }
   }
   if(maxamp > 0.000001){
         rescale = 1.0 / maxamp;
   }
   else {
         object_post((t_object *)x,
         "rescale: amplitude is too low to rescale: %f",maxamp);
         ATOMIC_DECREMENT(&b->b_inuse);
         return;
   }
   for(i = 0; i < b->b_frames * b->b_nchans; i++){
         b->b_samples[i] *= rescale;
   }
   object_method((t_object *)b, gensym("dirty"));
   ATOMIC_DECREMENT(&b->b_inuse);
}
```

Figure 7.11 Thread-safe buffer access

```
float *undo_samples; // contains samples to undo previous operation
long  undo_start; // start frame for the undo replace
long  undo_frames; // how many frames in the undo
long  can_undo; // flag that an undo is possible
long  undo_resize; // flag that the undo process will resize buffer
```

Figure 7.12 New components to support an undo method

```
void bed_normalize(t_bed *x)
{
// code omitted here
   // store samples for undo
   chunksize = b->b_frames * b->b_nchans * sizeof(float);
   if( x->undo_samples == NULL ){
         x->undo_samples = (float *) sysmem_newptr(chunksize);
   } else {
         x->undo_samples =
         (float *) sysmem_resizeptr(x->undo_samples, chunksize);
   }
   if(x->undo_samples == NULL){
         post("bed: cannot allocate memory for undo");
         x->can_undo = 0;
         ATOMIC_DECREMENT(&b->b_inuse);
         return;
   } else {
         x->can_undo = 1;
         x->undo_start = 0;
         x->undo_frames = b->b_frames;
         x->undo_resize = 0;
         sysmem_cpyptr(b->b_samples, x->undo_samples, chunksize);
   }
// code omitted here
}
```

Figure 7.13 Preparing the normalization method for an undo action

```
for( i  = 0; i < b->b_frames * b->b_nchans; i++){
    b->undo_samples[i] = b->b_samples[i];
}
```

Figure 7.14 Copying a memory block, one sample at a time

```
void bed_undo(t_bed *x)
{
   t_buffer *b;
   long chunksize; // size of memory alloc in bytes
   t_atom rv; // needed for message call

   if(! x->can_undo ){
         post("bed: nothing to undo");
         return;
   }
   if( ! attach_buffer(x) ){
         post("bed: %s is not a valid buffer",x->b_name->s_name);
         return;
   }
   b = x->buffy;
   ATOMIC_INCREMENT(&b->b_inuse);
    if (!b->b_valid) {
         ATOMIC_DECREMENT(&b->b_inuse);
         post("bed undo: not a valid buffer!");
         return;
   }
   chunksize = x->undo_frames * b->b_nchans * sizeof(float);
   if(x->undo_resize){
         ATOMIC_DECREMENT(&b->b_inuse);
         object_method_long(&b->b_obj, gensym("sizeinsamps"),
         x->undo_frames, &rv);
         ATOMIC_INCREMENT(&b->b_inuse);
   }
   sysmem_copyptr(x->undo_samples,
         b->b_samples + x->undo_start, chunksize);
   x->can_undo = 0;
   object_method((t_object *)b, gensym("dirty"));
   ATOMIC_DECREMENT(&b->b_inuse);
}
```

Figure 7.17 The undo method

```
void bed_fadein(t_bed *x, double fadetime)
{
   t_buffer *b;
   long chunksize; // size of memory alloc in bytes
   long fadeframes; // frames to fade for
   int i,j;
   if( ! attach_buffer(x) ){
         post("bed: %s is not a valid buffer",x->b_name->s_name);
         return;
   }
   b = x->buffy;
   ATOMIC_INCREMENT(&b->b_inuse);
      if (!b->b_valid) {
         ATOMIC_DECREMENT(&b->b_inuse);
         post("bed fade-in: not a valid buffer!");
               return;
   }
   fadeframes = fadetime * 0.001 * b->b_sr;
   if(fadetime <= 0 || fadeframes > b->b_frames){
         post("bed: bad fade time: %f", fadetime);
         ATOMIC_DECREMENT(&b->b_inuse);
         return;
   }
   chunksize = fadeframes * b->b_nchans * sizeof(float);
   if( x->undo_samples == NULL ){
         x->undo_samples = (float *) sysmem_newptr(chunksize);
   } else {
         x->undo_samples =
         (float *) sysmem_resizeptr(x->undo_samples, chunksize);
   }
   if(x->undo_samples == NULL){
         post("bed: cannot allocate memory for undo");
         x->can_undo = 0;
         ATOMIC_DECREMENT(&b->b_inuse);
         return;
   } else {
         x->can_undo = 1;
         x->undo_start = 0;
         x->undo_frames = fadeframes;
         x->undo_resize = 0;
         sysmem_copyptr(b->b_samples, x->undo_samples,
               chunksize);
   }
   for(i = 0; i < fadeframes; i++){
         for(j = 0; j < b->b_nchans; j++){
               b->b_samples[(i * b->b_nchans) + j]
                     *= (float)i / (float) fadeframes;
         }
   }
   object_method(&b->b_obj, gensym("dirty"));
   ATOMIC_DECREMENT(&b->b_inuse);
}
```

Figure 7.18 The fade-in method

```
long        undo_cut; // flag that last operation was a cut
```

Figure 7.19 The new “undo cut” flag component of the object structure

```
void bed_cut(t_bed *x, double start, double end)
{
   t_buffer *b;
   long chunksize; // size of memory alloc in bytes
   long cutframes; // frames to cut
   long startframe, endframe;
   t_atom rv; // return value, needed for message call
   long offset1, offset2;
   float *local_samples;
   long local_frames;

   if( ! attach_buffer(x) ){
         post("bed: %s is not a valid buffer",x->b_name->s_name);
         return;
   }
   b = x->buffy;
   ATOMIC_INCREMENT(&b->b_inuse);
         if (!b->b_valid) {
         ATOMIC_DECREMENT(&b->b_inuse);
         post("bed normalize: not a valid buffer!");
         return;
   }
   startframe = start * 0.001 * b->b_sr;
   endframe = end * 0.001 * b->b_sr;
   cutframes = endframe - startframe;
   if(cutframes <= 0 || cutframes > b->b_frames){
         post("bed: bad cut data: %f %f", start, end);
         ATOMIC_DECREMENT(&b->b_inuse);
         return;
   }
   x->undo_frames = cutframes;
   local_frames = b->b_frames;
   chunksize = local_frames * b->b_nchans * sizeof(float);
   local_samples = (float *) sysmem_newptr(chunksize);
   sysmem_copyptr(b->b_samples, local_samples,  chunksize);
   if(local_samples == NULL){
         post("bed: cannot store local samples");
         x->can_undo = 0;
         ATOMIC_DECREMENT(&b->b_inuse);
         return;
   }
   chunksize = cutframes * b->b_nchans * sizeof(float);
   if( x->undo_samples == NULL ){
         x->undo_samples = (float *) sysmem_newptr(chunksize);
   } else {
         x->undo_samples =
         (float *) sysmem_resizeptr(x->undo_samples, chunksize);
   }
```

Figure 7.20 The cut method

```
    if(x->undo_samples == NULL){
          post("bed: cannot allocate memory for undo");
          x->can_undo = 0;
          ATOMIC_DECREMENT(&b->b_inuse);
          return;
    } else {
          x->can_undo = 1;
          x->undo_start = startframe; // start frame
          x->undo_frames = cutframes;
          x->undo_resize = 1;
          sysmem_copyptr(b->b_samples + (startframe * b->b_nchans),
                x->undo_samples,  chunksize);
    }
    ATOMIC_DECREMENT(&b->b_inuse);
    object_method_long(&b->b_obj, gensym("sizeinsamps"),
          (b->b_frames - cutframes), &rv);
    ATOMIC_INCREMENT(&b->b_inuse);
    chunksize = startframe * b->b_nchans * sizeof(float);
    sysmem_copyptr(local_samples, b->b_samples,  chunksize);
    chunksize = (local_frames - endframe) *
          b->b_nchans * sizeof(float);
    offset1 = startframe * b->b_nchans;
    offset2 = endframe * b->b_nchans;
    sysmem_copyptr(local_samples + offset2,
          b->b_samples + offset1,  chunksize);
    object_method(&b->b_obj, gensym("dirty"));
    ATOMIC_DECREMENT(&b->b_inuse);
    sysmem_freeptr(local_samples);
    x->undo_cut = 1;
}
```

Figure 7.20 (continued)

```
int attach_any_buffer(t_buffer **b, t_symbol *b_name)
{
    t_object *o;
    o = b_name->s_thing;
    if(o == NULL){
          post("There is no object called %s",b_name->s_name);
          return 0;
    }
    if (ob_sym(o) == gensym("buffer~")) {
          *b = (t_buffer *) o;
          return 1;
    }
    else {
          post("%s is not a buffer", b_name->s_name);
          return 0;
    }
}
```

Figure 7.21 A utility function to attach any buffer

```
void bed_paste(t_bed *x, t_symbol *destname)
{
   t_buffer *destbuf = NULL;
   t_atom rv; // return value
   long chunksize;
   if(x->can_undo && x->undo_cut){
         if( ! attach_buffer(x) ){
               post("bed: %s is not a valid buffer",
               x->b_name->s_name);
               return;
         }
         if( attach_any_buffer(&destbuf, destname) ){
               if(destbuf->b_nchans != x->buffy->b_nchans){
                     post("bed: channel mismatch between %s and
                     %s",
                     destname->s_name, x->b_name->s_name);
                     return;
               }
               object_method_long(&destbuf->b_obj,
               gensym("sizeinsamps"), x->undo_frames, &rv);
               ATOMIC_INCREMENT(&destbuf->b_inuse);
               chunksize =
               x->undo_frames * destbuf->b_nchans * sizeof(float);
               sysmem_copyptr(x->undo_samples, destbuf->b_samples,
               chunksize);
               ATOMIC_DECREMENT(&destbuf->b_inuse);
         }
         else{
               post("bed: %s is not a valid buffer",
               destname->s_name);
         }
   }
   else {
         post("bed: nothing to paste");
   }

}
```

Figure 7.22 The paste method

```
if(x->undo_cut){
    local_frames = b->b_frames;
    chunksize = local_frames * b->b_nchans * sizeof(float);
    local_samples = (float *) sysmem_newptr(chunksize);
    sysmem_copyptr(b->b_samples, local_samples,  chunksize);
    ATOMIC_DECREMENT(&b->b_inuse);
    object_method_long(&b->b_obj, gensym("sizeinsamps"),
    x->undo_frames + local_frames, &rv);
    ATOMIC_INCREMENT(&b->b_inuse);
    chunksize = x->undo_start * b->b_nchans * sizeof(float);
    sysmem_copyptr(local_samples, b->b_samples, chunksize);
    chunksize = x->undo_frames * b->b_nchans * sizeof(float);
    offset = x->undo_start * b->b_nchans;
    sysmem_copyptr(x->undo_samples, b->b_samples + offset,
    chunksize);
    chunksize =
    (local_frames - x->undo_start) * b->b_nchans * sizeof(float);
    offset = (x->undo_start + x->undo_frames) * b->b_nchans;
    sysmem_copyptr(local_samples + (x->undo_start * b->b_nchans),
    b->b_samples + offset, chunksize);
    ATOMIC_DECREMENT(&b->b_inuse);
    x->undo_cut = 0;
    sysmem_freeptr(local_samples);
    return;
}
```

Figure 7.24 Enabling `bed_undo()` to reverse cut operations

```
void bed_dblclick(t_bed *x)
{
    t_buffer *b;
    if( ! attach_buffer(x) ){
          object_post(&b->b_obj, "%s is not a valid buffer",
          x->b_name->s_name);
          return;
    }
    b = x->buffy;
    object_method((t_object *)b, gensym("dblclick"));
}
```

Figure 7.25 Implementing double-click functionality

```
class_addmethod(c, (method)bed_dblclick, "dblclick", A_CANT, 0);
```

Figure 7.26 Binding `bed_dblclick()`

```
typedef struct _bed
{
      t_object    x_obj; // the Max object
      t_symbol    *b_name; // the name of the buffer
      t_garray    *buffy; // the Buffer
      long        b_valid; // state of the buffer
      long        b_frames; // frame count
      float       *b_samples; // samples
      float       b_1oversr; // 1 over the sampling rate
      float       *undo_samples; // samples to undo previous op
      long        undo_start; // start frame for the undo replace
      long        undo_frames; // how many frames in the undo
      long        can_undo; // flag that an undo is possible
      long        undo_resize; // flag undo will resize buffer
      long        undo_cut; // flag to un
      float       b_sr; // sampling rate
} t_bed;
```

Figure 7.27 The object structure for the Pd version of *bed*

```
void bed_setup(void)
{
   t_class *c;

   bed_class = class_new(gensym("bed"), (t_newmethod)bed_new,
   (t_method)bed_free, sizeof(t_bed), 0, A_SYMBOL, 0);
   c = bed_class;
   class_addmethod(c, (t_method)bed_info, gensym("info"),
         A_CANT, 0);
   class_addmethod(c, (t_method)bed_normalize,
         gensym("normalize"), 0);
   class_addmethod(c, (t_method)bed_fadein, gensym("fadein"),
               A_FLOAT, 0);
   class_addmethod(c, (t_method)bed_cut, gensym("cut"), A_FLOAT,
               A_FLOAT, 0);
   class_addmethod(c, (t_method)bed_paste, gensym("paste"),
               A_SYMBOL, 0);
   class_addmethod(c, (t_method)bed_undo,gensym("undo"),A_CANT,0);
   post("bed - from Designing Audio Objects for Max/MSP and Pd");
}
```

Figure 7.28 The initialization routine for *bed* in Pd

```
int attach_buffer(t_bed *x)
{
   t_garray *a;
   t_symbol *b_name;
   float *b_samples;
   int b_frames;
   b_name = x->b_name;
   x->b_valid = 0;
   if (!(a = (t_garray *)pd_findbyclass(b_name, garray_class))) {
         if (b_name->s_name){
               pd_error(x, "bed: %s: no such array",b_name->s_name);
         }
         return x->b_valid;
    }
   if (!garray_getfloatarray(a, &b_frames, &b_samples)) {
         pd_error(x, "bed: bad array for %s", b_name->s_name);
         return x->b_valid;
   }
   else  {
         x->b_valid = 1;
         x->b_frames = (long)b_frames;
         x->b_samples = b_samples;
         x->b_sr = sys_getsr();
         if(x->b_sr <= 0){
               x->b_sr = 44100.0;
         }
         x->b_1oversr = 1.0 / x->b_sr;
         x->buffy = a;
   }
   return x->b_valid;
}
```

Figure 7.29 The Pd version of `attach_buffer()`

```
void bed_undo(t_bed *x)
{
   t_garray *a;
   long chunksize; // size of memory alloc in bytes
   float *local_samples; // for undoing a cut
   long local_frames;
   long offset;
   long oldsize; // Pd bookkeeping
   if(! x->can_undo ){
         post("bed: nothing to undo");
         return;
   }
   if( ! attach_buffer(x) ){
         post("bed: %s is not a valid buffer",x->b_name->s_name);
         return;
   }
   a = x->buffy;
   // take care of special case for undo cut
   if(x->undo_cut){
         // copy everything to local buffer
         local_frames = x->b_frames;
         chunksize = local_frames * sizeof(float);
         local_samples = getbytes(chunksize);
         memcpy(local_samples, x->b_samples, chunksize);
         // now resize buffer to incorporate cut segment
         garray_resize(a, x->undo_frames + local_frames);
         // copy first part of buffer back
         chunksize = x->undo_start * sizeof(float);
         memcpy(x->b_samples, local_samples, chunksize);
         // now copy the cut piece back in
         chunksize = x->undo_frames * sizeof(float);
         memcpy(x->b_samples + x->undo_start, x->undo_samples,
                     chunksize);
         // finally, add the last piece from the original
         chunksize = (local_frames - x->undo_start) *
               sizeof(float);
         offset = x->undo_start + x->undo_frames;
         memcpy(x->b_samples + offset,
               local_samples + x->undo_start,  chunksize);
         x->undo_cut = 0;
         oldsize = local_frames * sizeof(float);
         freebytes(local_samples, oldsize);
         garray_redraw(a);
         return;
   }
   chunksize = x->undo_frames * sizeof(float);
   if(x->undo_resize){
         garray_resize(a, x->undo_frames);
   }
   // copy old samples back into (possibly resized) buffer
   memcpy(x->b_samples + x->undo_start, x->undo_samples,
         chunksize);
   // now nothing left to undo
   x->can_undo = 0;
   garray_redraw(a);
}
```

Figure 7.30 The undo method for *bed* in Pd

```
void bed_cut(t_bed *x, t_floatarg start, t_floatarg end)
{
    t_garray *a;
    long chunksize; // size of memory alloc in bytes
    long oldsize; // Pd bookkeeping
    long cutframes; // frames to cut
    long startframe, endframe;
    float *local_samples;
    long local_frames;
    if( ! attach_buffer(x) ){
          post("bed: %s is not a valid buffer",x->b_name->s_name);
          return;
    }
    a = x->buffy;
    startframe = start * 0.001 * x->b_sr;
    endframe = end * 0.001 * x->b_sr;
    cutframes = endframe - startframe;
    if(cutframes <= 0 || cutframes > x->b_frames){
          post("bed: bad cut data: %f %f", start, end);
          return;
    }
    x->undo_frames = cutframes;
    local_frames = x->b_frames;
    // store samples for undo (copy everything)
    chunksize = local_frames * sizeof(float); // all arrays are mono
    local_samples = getbytes(chunksize); // use Pd memory function
    memcpy(local_samples, x->b_samples, chunksize); // C lib function
    chunksize = cutframes * sizeof(float);
    if( x->undo_samples == NULL ){
          x->undo_samples = getbytes(chunksize);
    } else {
          oldsize = x->undo_frames * sizeof(float);
          x->undo_samples = resizebytes(x->undo_samples, oldsize,
                chunksize);
    }
    if(x->undo_samples == NULL){
          post("bed: cannot allocate memory for undo");
          x->can_undo = 0;
          return;
    } else {
          x->can_undo = 1;
          x->undo_start = startframe; // start frame
          x->undo_frames = cutframes;
          x->undo_resize = 1;
          memcpy(x->undo_samples, x->b_samples + startframe, chunksize);
    }
    garray_resize(a, (x->b_frames - cutframes));
    // copy up to the start of the cut
    chunksize = startframe * sizeof(float);
    memcpy(x->b_samples, local_samples, chunksize);
    chunksize = (local_frames - endframe) * sizeof(float);
    memcpy(x->b_samples + startframe, local_samples + endframe,
          chunksize);
    // free local memory
    oldsize = local_frames * sizeof(float);
    freebytes(local_samples, oldsize);
    x->undo_cut = 1;
    garray_redraw(x->buffy);
}
```

Figure 7.31 The Pd version of `bed_cut()`

```
int attach_any_buffer(t_garray **dest_array, t_symbol *b_name)
{
   t_garray *a;
   int b_valid = 0;
   if (!(a = (t_garray *)pd_findbyclass(b_name, garray_class))) {
         if (b_name->s_name) post("ben: %s: no such array",
         b_name->s_name);
         return 0;
    } else {
         b_valid = 1;
   }
   *dest_array = a;
   return b_valid;
}
```

Figure 7.32 The Pd version of `attach_any_buffer()`

```
void bed_paste(t_bed *x, t_symbol *destname)
{
    t_garray *a;
    t_garray *destbuf = NULL;
    long chunksize;
    int destbuf_b_frames;
    float *destbuf_b_samples;
    if(x->can_undo){
         if( ! attach_buffer(x) ){
               post("bed: %s is not a valid buffer",
               x->b_name->s_name);
               return;
         }
         if( attach_any_buffer(&destbuf, destname) ){
               if (!garray_getfloatarray(destbuf, &destbuf_b_frames,
                     &destbuf_b_samples)) {
                     pd_error(x, "bed: bad array for %s",
                           destname->s_name);
                     return;
               }
               garray_resize(destbuf, x->undo_frames);
               chunksize = x->undo_frames * sizeof(float);
               if (!garray_getfloatarray(destbuf, &destbuf_b_frames,
                     &destbuf_b_samples)) {
                     pd_error(x, "bed: bad array for %s",
                     destname->s_name);
                     return;
               }
               memcpy(destbuf_b_samples,x->undo_samples,chunksize);
               garray_redraw(destbuf);
         }
         else{
               post("bed: %s is not a valid buffer",
               destname->s_name);
         }
    } else {
         post("bed: nothing to paste");
    }
}
```

Figure 7.33 The Pd version of `bed_paste()`

```
t_int *cleaner_perform(t_int *w)
{
    t_float *input = (t_float *) (w[1]);
    t_float *threshmult = (t_float *) (w[2]);
    t_float *multiplier = (t_float *) (w[3]);
    t_float *output = (t_float *) (w[4]);
    t_int n = w[5];
    float maxamp = 0.0;
    float threshold;
    float mult;
    int i;
    for(i = 0; i < n; i++){
          if(maxamp < input[i]){
                maxamp = input[i];
          }
    }
    threshold = *threshmult * maxamp;
    mult = *multiplier;
    for(i = 0; i < n; i++){
          if(input[i] < threshold){
                input[i] *= mult;
          }
          output[i] = input[i];
    }
    return w + 6;
}
```

Figure 8.2 Adaptive noise reduction coded in C

```
while(phase > PI){
    phase -= TWOPI;
}
while(phase < -PI){
    phase += TWOPI;
}
```

Figure 8.8 Implementing phase wrapping in C code

```
#define SCRUBBER_EMPTY 0
#define SCRUBBER_FULL 1
```

Figure 8.10 *scrubber~* buffer state constants

```
typedef struct _scrubber {
     t_pxobject obj; // Max/MSP proxy object
     float **amplitudes; // contains spectral frames
     float **phases; // contains spectral frames
     float duration_ms; // duration in milliseconds
     long recorded_frames; // counter for recording of frames
     long framecount; // total frames in spectrum
     long oldframes; // for resizing memory
     long fftsize; // number of bins in a frame
     float sr; // sampling rate
     float frame_position; // current frame
     float increment; // speed to advance through the spectrum
     short acquire_sample; //flag to begin sampling
     float sync; // location in buffer (playback or recording)
     float overlap;// overlap factor for STFT
     short buffer_status; // empty or full
     float last_position; //last spectrum position (scaled 0-1)
} t_scrubber;
```

Figure 8.11 The *scrubber~* object structure

```
void scrubber_init_memory(t_scrubber *x)
{
   long framecount = x->framecount;
   long oldframes = x->oldframes;
   long fftsize = x->fftsize;
   long framesize = fftsize / 2;
   long bytesize;
   int i;
   if(framecount <= 0){
         post("scrubber~: bad frame count: %d", framecount);
         return;
   }
   if(fftsize <= 0){
         post("scrubber~: bad size: %d", fftsize);
         return;
   }
   x->buffer_status = SCRUBBER_EMPTY;
   if(x->amplitudes == NULL){
         bytesize = framecount * sizeof(float *);
         x->amplitudes = (float **) sysmem_newptr(bytesize);
         x->phases = (float **) sysmem_newptr(bytesize);
         bytesize = framesize * sizeof(float);
         for(i = 0; i < framecount; i++){
               x->amplitudes[i] = (float *)
                     sysmem_newptr(bytesize);
               x->phases[i] = (float *) sysmem_newptr(bytesize);
         }
   }
   else {
         for(i = 0; i < oldframes; i++){
               sysmem_freeptr(x->amplitudes[i]);
               sysmem_freeptr(x->phases[i]);
         }
         bytesize = framecount * sizeof(float *);
         x->amplitudes =
               (float **)sysmem_resizeptr(x->amplitudes,
         bytesize);
         x->phases =
               (float **)sysmem_resizeptr(x->phases, bytesize);
         bytesize = framesize * sizeof(float);
         for(i = 0; i < framecount; i++){
               x->amplitudes[i] = (float *)
                     sysmem_newptr(bytesize);
               x->phases[i] = (float *)sysmem_newptr(bytesize);
         }
         x->oldframes = framecount;
   }
}
```

Figure 8.12 The memory allocation routine

```
void *scrubber_new(t_symbol *s, int argc, t_atom *argv)
{
   t_scrubber *x = (t_scrubber *)object_alloc(scrubber_class);
   dsp_setup((t_pxobject *)x, 4);
   outlet_new((t_object *)x, "signal");
   outlet_new((t_object *)x, "signal");
   outlet_new((t_object *)x, "signal");
   x->amplitudes = NULL;
   x->framecount = 1; // one frame for initialization purposes
   x->acquire_sample = 0;
   x->fftsize = 1024;
   x->frame_position = 0;
   x->oldframes = 0;
   x->duration_ms = 5000.0; // default size
   x->buffer_status = SCRUBBER_EMPTY;
   if( argc >= 1){
         x->duration_ms = atom_getfloatarg(0, argc, argv);
   }
   x->last_position = 0;
   x->overlap = 8.0;
   x->sr = 44100.0;
   scrubber_init_memory(x);
   return x;
}
```

Figure 8.13 The new instance routine

```
t_int *scrubber_perform(t_int *w)
{
   t_scrubber *x = (t_scrubber *) (w[1]);
   t_float *mag_in = (t_float *) (w[2]);
   t_float *phase_in = (t_float *) (w[3]);
   t_float *increment = (t_float *) (w[4]);
   t_float *position = (t_float *) (w[5]);
   t_float *mag_out = (t_float *) (w[6]);
   t_float *phase_out = (t_float *) (w[7]);
   t_float *sync = (t_float *) (w[8]);
   t_int n = w[9];
   long framecount = x->framecount;
   long recorded_frames = x->recorded_frames;
   float frame_position = x->frame_position;
   float **amplitudes = x->amplitudes;
   float **phases = x->phases;
   int i;
   long iframe_position;
   float sync_val;
   short acquire_sample = x->acquire_sample;
   float last_position = x->last_position;

```

Figure 8.14 The *scrubber~* perform routine

```
    if(acquire_sample){
        sync_val = (float) recorded_frames / (float) framecount;
        for (i = 0; i < n; i++) {
            amplitudes[recorded_frames][i] = mag_in[i];
            phases[recorded_frames][i] = phase_in[i];
        }
        for(i = 0; i < n; i++){
            // send silence while sampling
            mag_out[i] = 0.;
            phase_out[i] = 0.;
            sync[i] = sync_val;
        }
        ++recorded_frames;
        if(recorded_frames >= framecount){
            acquire_sample = 0;
            x->buffer_status = SCRUBBER_FULL;
        }
    }
    else if(x->buffer_status == SCRUBBER_FULL) {
        sync_val = frame_position / (float) framecount;
        if(last_position != *position && *position >= 0.0
            && *position <= 1.0){
            last_position = *position;
            frame_position = last_position *
                (float)(framecount - 1);
        }
        frame_position += *increment;

        while(frame_position < 0.){
            frame_position += framecount;
        }
        while(frame_position >= framecount){
            frame_position -= framecount;
        }
        iframe_position = floor(frame_position);

        for (i = 0; i < n; i++) {
            mag_out[i] = amplitudes[iframe_position][i];
            phase_out[i] = phases[iframe_position][i];
            sync[i] = sync_val;
        }
    }
    else {
        for(i = 0; i < n; i++){
            mag_out[i] = 0.0;
            phase_out[i] = 0.0;
            sync[i] = 0.0;
        }
    }
    x->last_position = last_position;
    x->frame_position = frame_position;
    x->acquire_sample = acquire_sample;
    x->recorded_frames = recorded_frames;
    return w + 10;
}
```

Figure 8.14 (continued)

```
void scrubber_sample(t_scrubber *x)
{
    x->acquire_sample = 1;
    x->recorded_frames = 0;
    x->buffer_status = SCRUBBER_EMPTY;
}
```

Figure 8.15 Initiating the sampling process

```
void scrubber_dsp(t_scrubber *x, t_signal **sp, short *count)
{
   long blocksize = sp[0]->s_n;
   float local_sr = sp[0]->s_sr;
   long local_fftsize = blocksize * 2;
   float framedur;
   long new_framecount;
   if(!local_sr){
         return;
   }
   framedur = local_fftsize / x->sr;
   new_framecount = 0.001 * x->duration_ms *
         x->overlap / framedur;
   if(x->fftsize != local_fftsize || x->sr != sp[0]->s_sr ||
               x->framecount != new_framecount){
         x->fftsize = local_fftsize;
         x->sr = sp[0]->s_sr;
         x->framecount = new_framecount;
         scrubber_init_memory(x);
   }
   dsp_add(scrubber_perform, 9, x,  sp[0]->s_vec, sp[1]->s_vec,
         sp[2]->s_vec, sp[3]->s_vec, sp[4]->s_vec, sp[5]->s_vec,
         sp[6]->s_vec, sp[0]->s_n);
}
```

Figure 8.16 The dsp method

```
void scrubber_overlap(t_scrubber *x, t_symbol *msg, short argc,
    t_atom *argv)
{
    float old_overlap = x->overlap;
    if(argc >= 1){
          x->overlap = atom_getfloatarg(0,argc,argv);
          if(x->overlap <= 0){
                post("scrubber~: bad overlap: %f", x->overlap);
                x->overlap = old_overlap;
                return;
          }
          if(x->overlap != old_overlap){
                scrubber_init_memory(x);
          }
    }
}
```

Figure 8.17 The overlap method

```
void scrubber_resize(t_scrubber *x, t_symbol *msg, short argc,
    t_atom *argv)
{
    float old_size = x->duration_ms;
    float framedur;
    if(argc >= 1){
          x->duration_ms = atom_getfloatarg(0,argc,argv);
    }
    else {
          return;
    }
    if(old_size == x->duration_ms){
          return;
    }
    if(x->duration_ms > 0.0 && x->sr > 0.0 && x->fftsize > 0.0 &&
          x->overlap > 0.0){
          framedur = 2.0 * x->fftsize / x->sr;
          x->oldframes = x->framecount;
          x->framecount = 0.001 * x->duration_ms *
                x->overlap / framedur;
          scrubber_init_memory(x);
    }
}
```

Figure 8.18 Resizing the spectral buffer

```
void scrubber_free(t_scrubber *x)
{
   int i;
   dsp_free(&x->obj);
   if(x->amplitudes != NULL){
         for(i = 0; i < x->framecount; i++){
               sysmem_freeptr(x->amplitudes[i]);
               sysmem_freeptr(x->phases[i]);
         }
         sysmem_freeptr(x->amplitudes);
         sysmem_freeptr(x->phases);
   }
}
```

Figure 8.19 The `scrubber_free()` routine

```
void windowvec_dsp(t_windowvec *x, t_signal **sp, short *count)
{
   int i;
   float twopi = 8. * atan(1);
   if(x->vecsize != sp[0]->s_n){
         x->vecsize = sp[0]->s_n;
         if(x->envelope == NULL){
               x->envelope = (float *)
               getbytes(x->vecsize * sizeof(float));
         } else {
               x->envelope = (float *)
               resizebytes(x->envelope, x->oldbytes,
                     x->vecsize * sizeof(float));
         }
         x->oldbytes = x->vecsize * sizeof(float);
         for(i = 0 ; i < x->vecsize; i++){
               x->envelope[i] =
               -0.5 * cos(twopi * (i / (float)x->vecsize)) + 0.5;
         }
   }
   dsp_add(windowvec_perform, 4, x, sp[0]->s_vec,
         sp[1]->s_vec, sp[0]->s_n);
}
```

Figure 8.21 The dsp routine for *windowvec~*

```
t_int *windowvec_perform(t_int *w)
{
   t_windowvec *x = (t_windowvec *) (w[1]);
   t_float *input = (t_float *) (w[2]);
   t_float *output = (t_float *) (w[3]);
   t_int n = w[4];
   int i;
   float *envelope = x->envelope;
   for(i=0; i < n; i++){
         output[i] = input[i] * envelope[i];
   }
   return w + 5;
}
```

Figure 8.22 The perform routine for *windowvec~*

```
t_scrubber *x = (t_scrubber *) (w[1]);
t_float *real_in = (t_float *) (w[2]); // real
t_float *imag_in = (t_float *) (w[3]); // imag
t_float *increment = (t_float *) (w[4]);
t_float *position = (t_float *) (w[5]);
t_float *real_out = (t_float *) (w[6]);
t_float *imag_out = (t_float *) (w[7]);
t_float *sync = (t_float *) (w[8]);
t_int n = w[9];
long framecount = x->framecount;
long recorded_frames = x->recorded_frames;
float frame_position = x->frame_position;
float **amplitudes = x->amplitudes;
float **phasediffs = x->phasediffs;
int i;
float real, imag;
long iframe_position;
float sync_val;
short acquire_sample = x->acquire_sample;
float last_position = x->last_position;
float *lastphase_in = x->lastphase_in;
float *lastphase_out = x->lastphase_out;
float phase_out, mag_out;
float local_phase, phasediff;
int N2 = n / 2; // half of FFT size
```

Figure 8.23 Inputs to the perform routine

```
if(acquire_sample){
   sync_val = (float) recorded_frames / (float) framecount;
   for (i = 0; i < N2 + 1; i++) {
         real = real_in[i];
         imag = (i == 0 || i == N2 ? 0. : imag_in[i]);
         amplitudes[recorded_frames][i] = hypot(real, imag);
         local_phase = -atan2(imag, real);
         phasediff = local_phase - lastphase_in[i];
         lastphase_in[i] = local_phase;
         while(phasediff > PI){
               phasediff -= TWOPI;
         }
         while(phasediff < -PI){
               phasediff += TWOPI;
         }
         phasediffs[recorded_frames][i] = phasediff;
   }
   for(i = 0; i < n; i++){
         real_out[i] = 0.;
         imag_out[i] = 0.;
         sync[i] = sync_val;
   }
   ++recorded_frames;
   if(recorded_frames >= framecount){
         acquire_sample = 0;
         x->buffer_status = SCRUBBER_FULL;
   }
}
```

Figure 8.24 The analysis loops

```
else if(x->buffer_status == SCRUBBER_FULL) {
   sync_val = frame_position / (float) framecount;
   if( last_position != *position && *position >= 0.0
         && *position <= 1.0 ){
         last_position = *position;
         frame_position = last_position * (float)(framecount - 1);
   }
   frame_position += *increment;
   while(frame_position < 0.){
         frame_position += framecount;
   }
   while(frame_position >= framecount){
         frame_position -= framecount;
   }
   iframe_position = floor(frame_position);
   for ( i = 0; i < N2+1; i++ ) {
         mag_out = amplitudes[iframe_position][i];
         lastphase_out[i] += phasediffs[iframe_position][i];
         local_phase = lastphase_out[i];
         real_out[i] = mag_out * cos(local_phase);
         imag_out[i] = (i == 0 || i == N2) ? 0.0 : -mag_out *
               sin(local_phase);
         sync[i] = sync_val;
   }
   for ( i = N2+1; i < n; i++ )
   {
         real_out[i] = 0.;
         imag_out[i] = 0.;
         sync[i] = sync_val;
   }
}
```

Figure 8.25 The *scrubber~* resynthesis block

```
void scrubber_dsp(t_scrubber *x, t_signal **sp, short *count)
{
   float local_sr = sys_getsr();
   long local_blocksize = sp[0]->s_n;
   float framedur;
   long new_framecount;
   if(!local_sr){
         return;
   }
   framedur = local_blocksize / x->sr;
   new_framecount = 0.001 * x->duration_ms *
         x->overlap / framedur;
   if(x->fftsize != local_blocksize || x->sr != local_sr||
               x->framecount != new_framecount) {
         x->fftsize = local_blocksize;
         x->sr = local_sr;
         x->framecount = new_framecount;
         scrubber_init_memory(x);
   }
   dsp_add(scrubber_perform, 9, x,  sp[0]->s_vec, sp[1]->s_vec,
         sp[2]->s_vec, sp[3]->s_vec, sp[4]->s_vec,
         sp[5]->s_vec, sp[6]->s_vec, sp[0]->s_n);
}
```

Figure 8.26 The *scrubber~* dsp method for Pd

```
void scrubber_init_memory(t_scrubber *x)
{
   long framecount = x->framecount;
   long oldframes = x->oldframes;
   long fftsize = x->fftsize;
   long fftsize2 = fftsize / 2;
   int i;
   long bytesize;
   if(framecount <= 0){
         post("bad frame count: %d", framecount);
         return;
   }
   if(fftsize <= 0){
         post("bad size: %d", fftsize);
         return;
   }
   x->buffer_status = SCRUBBER_EMPTY;
   bytesize = framecount * sizeof(float *);
   if(x->amplitudes == NULL){
         x->amplitudes = (float **) malloc(bytesize);
         x->phasediffs = (float **) malloc(bytesize);
         bytesize = (fftsize2 + 1) * sizeof(float);
         for(i = 0; i < framecount; i++){
               x->amplitudes[i] = (float *) malloc(bytesize);
               x->phasediffs[i] = (float *) malloc(bytesize);
         }
         x->lastphase_in = (float *) malloc(bytesize);
         x->lastphase_out = (float *) malloc(bytesize);
         memset(x->lastphase_in, 0, bytesize);
         memset(x->lastphase_out, 0, bytesize);
   }
   else {
         for(i = 0; i < oldframes; i++){
               free(x->amplitudes[i]);
               free(x->phasediffs[i]);
         }
         bytesize = framecount * sizeof(float *);
         x->amplitudes = (float **)realloc(x->amplitudes,
bytesize);
         x->phasediffs = (float **)realloc(x->phasediffs,
bytesize);
         bytesize = (fftsize2 + 1) * sizeof(float);
         for(i = 0; i < framecount; i++){
               x->amplitudes[i] = (float*)malloc(bytesize);
               x->phasediffs[i] = (float*)malloc(bytesize);
         }
         x->lastphase_in =
               (float *) realloc(x->lastphase_in, bytesize);
         x->lastphase_out =
               (float *) realloc(x->lastphase_out, bytesize);
         memset(x->lastphase_in, 0, bytesize);
         memset(x->lastphase_out, 0, bytesize);
   }
}
```

Figure 8.27 Coding memory storage for Pd using standard C library functions

```
void scrubber_free(t_scrubber *x)
{
   int i;
   if(x->amplitudes != NULL){
         for(i = 0; i < x->framecount; i++){
               free(x->amplitudes[i]);
               free(x->phasediffs[i]);
         }
         free(x->amplitudes);
         free(x->phasediffs);
         free(x->lastphase_in);
         free(x->lastphase_out);
   }
}
```

Figure 8.28 The free function using standard C functions

```
t_int *vpdelay_perform(t_int *w)
{
    t_vpdelay *x = (t_vpdelay *) (w[1]);
    t_float *input = (t_float *) (w[2]);
    t_float *delaytime = (t_float *) (w[3]);
    t_float *feedback = (t_float *) (w[4]);
    t_float *output = (t_float *) (w[5]);
    t_int n = w[6];
    float sr = x->sr;
    float *delay_line = x->delay_line;
    float *read_ptr  = x->read_ptr;
    float *write_ptr = x->write_ptr;
    long delay_length = x->delay_length;
    float *endmem = delay_line + delay_length;
    short delaytime_connected = x->delaytime_connected;
    short feedback_connected = x->feedback_connected;
    float delaytime_float = x->delay_time;
    float feedback_float = x->feedback;
    float fraction;
    float fdelay;
    float samp1, samp2;
    long idelay;
    float srms = sr / 1000.0;
    float out_sample, feedback_sample;
    while(n--){
          if(delaytime_connected){
                fdelay = *delaytime++ * srms;
          }
          else {
                fdelay = delaytime_float * srms;
          }
          while(fdelay > delay_length){
                fdelay -= delay_length;
          }
          while(fdelay < 0){
                fdelay += delay_length;
          }
          idelay = trunc(fdelay);
          fraction = fdelay - idelay;
          read_ptr = write_ptr - idelay;
          while(read_ptr < delay_line){
                read_ptr += delay_length;
          }
          samp1 = *read_ptr++;
          if(read_ptr == endmem){
                read_ptr = delay_line;
          }
          samp2 = *read_ptr;
          out_sample = samp1 + fraction * (samp2-samp1);
```

Figure 9.1 A perform routine for *vpdelay~* using pointers to access the delay line

```
        if(feedback_connected) {
            feedback_sample = out_sample * *feedback++;
        }
        else {
            feedback_sample = out_sample * feedback_float;
        }
        if(fabs(feedback_sample) < 0.0000001){
            feedback_sample = 0.0;
        }
        *write_ptr++ = *input++ + feedback_sample;
        *output++ = out_sample;
        if(write_ptr == endmem){
            write_ptr = delay_line;
        }
    }
    x->write_ptr = write_ptr;
    return w + 7;
}
```

Figure 9.1 (continued)

```
x->write_ptr = x->delay_line;
```

Figure 9.2 Assigning the write pointer to the start of the delay line

```
var ocnt = 200;
var p = this.patcher;
function build()
{
    var i;
    var h = 20, v = 20;
    var source = p.newdefault(h,v,"cycle~","440");
    var newdel;
    v += 30;
    for(i = 0; i < ocnt; i++){
        newdel = p.newdefault(h,v,"vpdelay~", 10, 10, 0.5);
        p.connect(source, 0, newdel, 0);
    }
}
```

Figure 9.3 JavaScript code for benchmarking *vpdelay~*

```
      if(delaytime_connected && feedback_connected){
// DSP loop for both connected
      }
      else if(delaytime_connected){
// DSP loop for only delay time connected
      }
      else if(feedback_connected){
// DSP loop for feedback connected
      }
      else {
// DSP loop for neither connected.
      }
```

Figure 9.4 Branching structure to select most efficient processing scheme

```
else {
   fdelay = delaytime_float * srms;
   while(fdelay > delay_length){
         fdelay -= delay_length;
   }
   while(fdelay < 0){
         fdelay += delay_length;
   }
   idelay = trunc(fdelay);
   fraction = fdelay - idelay;
   while(n--){
         read_ptr = write_ptr - idelay;
         while(read_ptr < delay_line){
               read_ptr += delay_length;
         }
         samp1 = *read_ptr++;
         if(read_ptr == endmem){
               read_ptr = delay_line;
         }
         samp2 = *read_ptr;
         out_sample = samp1 + fraction * (samp2-samp1);
         feedback_sample = out_sample * feedback_float;
         if(fabs(feedback_sample) < 0.0000001){
               feedback_sample = 0.0;
         }
         *write_ptr++ = *input++ + feedback_sample;
         *output++ = out_sample;
         if(write_ptr == endmem){
               write_ptr = delay_line;
         }
   }
}
```

Figure 9.5 The DSP branch where neither delay time nor feedback input is a signal

```
function build_both_connected()
{
   var i;
   var h = 20, v = 20;
   var source = p.newdefault(h,v,"cycle~", "440");
   var sdel = p.newdefault(h+100,v, "sig~", 2.0);
   var sfeed= p.newdefault(h+200,v, "sig~", 0.5);
   var newdel;
   v += 30;
   for(i = 0; i < ocnt; i++){
         newdel = p.newdefault(h,v,"vpdelay~", 10, 10, 0.5);
         p.connect(source, 0, newdel, 0);
         p.connect(sdel,0, newdel,1);
         p.connect(sfeed,0, newdel,2);
   }
}
```

Figure 9.6 Benchmarking *vpdelay~* with signal input for both feedback and delay time

```
idelay = (int) fdelay;
```

Figure 9.10 Replacing the function call `trunc()`

```
FIX_DENORM_FLOAT(feedback_sample);
```

Figure 9.11 Calling the Max/MSP macro to fix denormal numbers

```
x->sr = sys_getsr();
// code omitted
x->delay_length = x->sr * x->maximum_delay_time + 1;
x->delay_bytes = x->delay_length * sizeof(float);
x->delay_line = (float *) sysmem_newptrclear(x->delay_bytes);
// code omitted
x->write_ptr = x->delay_line;
```

Figure 9.13 Dangerous initialization code

```
if(x->sr != sp[0]->s_sr){
   x->sr = sp[0]->s_sr;
   x->delay_length = x->sr * x->maximum_delay_time + 1;
   x->delay_bytes = x->delay_length * sizeof(float);
   x->delay_line = (float *)
         sysmem_resizeptrclear((void *)x->delay_line,
         x->delay_bytes);
         if(x->delay_line == NULL){
               error("vpdelay~: cannot realloc %d bytes of memory",
                     x->delay_bytes);
               return;
         }
}
```

Figure 9.14 A problem with pointers in the dsp method for *vpdelay~*

```
void *vpdelay_new(t_symbol *s, short argc, t_atom *argv)
{
   float delmax = 100.0, deltime = 100.0, feedback = 0.1;
   t_vpdelay *x = object_alloc(vpdelay_class);
   dsp_setup(&x->obj, 3);
   outlet_new((t_object *)x, "signal");
   atom_arg_getfloat(&delmax, 0, argc, argv);
   atom_arg_getfloat(&deltime, 1, argc, argv);
   atom_arg_getfloat(&feedback, 2, argc, argv);
   if(delmax <= 0){
         delmax = 250.0;
   }
   x->maximum_delay_time = delmax * 0.001;
   x->delay_time = deltime;
   if(x->delay_time > delmax || x->delay_time <= 0.0){
         error("illegal delay time: %f, reset to 1 ms",
               x->delay_time);
         x->delay_time = 1.0;
   }
   x->sr = 0.0;
   x->feedback = feedback;
   return x;
}
```

Figure 9.15 The revised new instance routine for *vpdelay~*

```
void vpdelay_dsp(t_vpdelay *x, t_signal **sp, short *count)
{
   if(!sp[0]->s_sr){
         return;
   }
   x->delaytime_connected = count[1];
   x->feedback_connected = count[2];
   if(x->sr != sp[0]->s_sr){
         x->sr = sp[0]->s_sr;
         x->delay_length = x->sr * x->maximum_delay_time + 1;
         x->delay_bytes = x->delay_length * sizeof(float);
         if(x->delay_line == NULL){
               x->delay_line =
                     (float *) sysmem_newptrclear(x->delay_bytes);
         }
         else {
               x->delay_line = (float *)
                     sysmem_resizeptrclear((void *)x->delay_line,
                     x->delay_bytes);
         }
         if(x->delay_line == NULL){
               error("vpdelay~: cannot realloc %d bytes of
memory",
                     x->delay_bytes);
               return;
         }
         x->write_ptr = x->delay_line;
   }
   dsp_add(vpdelay_perform, 6, x, sp[0]->s_vec, sp[1]->s_vec,
         sp[2]->s_vec, sp[3]->s_vec, sp[0]->s_n);
}
```

Figure 9.16 Setting the write pointer and memory addresses in the dsp method

```
t_class *c = oscil_class = class_new("oscil_attributes~",
  (method)oscil_new,(method)oscil_free, sizeof(t_oscil),
0,A_GIMME,0);
```

Figure 10.1 Using a temporary class variable for more compact code

```
CLASS_ATTR_FLOAT(c, "frequency", 0, t_oscil, a_frequency);
```

Figure 10.2 Creating a float attribute for frequency

```
x->a_frequency = 440.0;
attr_args_process(x, argc, argv);
```

Figure 10.3 Processing the frequency attribute in the new instance routine

```
t_max_err a_frequency_set(t_oscil *x, void *attr, long ac,
    t_atom *av)
{
    if (ac&&av) {
          x->a_frequency = atom_getfloat(av);
          x->si = x->a_frequency * x->si_factor;
    }
    return MAX_ERR_NONE;
}
```

Figure 10.5 Overriding the `a_frequency_set()` method

```
CLASS_ATTR_ACCESSORS(c, "frequency", NULL, a_frequency_set);
```

Figure 10.6 Binding the frequency message in the initialization routine

```
long a_xfadetype;
```

Figure 10.8 The object structure component for a fadetype attribute

```
CLASS_ATTR_LONG(c, "xfade", 0, t_oscil, a_xfadetype);
CLASS_ATTR_LABEL(c, "xfade", 0, "Crossfade");
CLASS_ATTR_ENUMINDEX(c, "xfade", 0, "\"No Fade\" \"Linear Fade\"
\"Equal Power Fade\"");
```

Figure 10.9 Attribute macro calls to define the `xfade` attribute and its behavior

```
long xfadetype = x->a_xfadetype;
```

Figure 10.10 Dereferencing the fadetype attribute in the perform routine

```
t_symbol *a_waveform;
```

Figure 10.11 The waveform attribute component

```
CLASS_ATTR_SYM(c, "waveform", 0, t_oscil, a_waveform);
CLASS_ATTR_LABEL(c, "waveform", 0, "Oscillator Waveform");
CLASS_ATTR_ENUM(c, "waveform", 0, "Sine Triangle Square Sawtooth
Pulse Additive");
CLASS_ATTR_ACCESSORS(c, "waveform", NULL, a_waveform_set);
```

Figure 10.12 Defining the `waveform` attribute

```
t_max_err a_waveform_set(t_oscil *x, void *attr, long ac,
   t_atom *av)
{
   if (av) {
         x->a_waveform = atom_getsym(av);
         if (x->a_waveform == gensym("Sine")) {
               oscil_sine(x);
         } else if (x->a_waveform == gensym("Triangle")) {
               oscil_triangle(x);
         } else if (x->a_waveform == gensym("Square")) {
               oscil_square(x);
         } else if (x->a_waveform == gensym("Sawtooth")) {
               oscil_sawtooth(x);
         } else if (x->a_waveform == gensym("Pulse")) {
               oscil_pulse(x);
         } else if(x->a_waveform == gensym("Additive")){
               oscil_additive(x);
         }
         else {
               error("%s is not a legal waveform",
                     x->waveform->s_name);
               oscil_sine(x);
         }
   }
   return MAX_ERR_NONE;
}
```

Figure 10.13 The waveform setter method

```
t_float a_amplitudes[8];
```

Figure 10.14 Defining an attribute array with static memory

```
t_float *a_amplitudes;
```

Figure 10.15 The amplitude array component

```
CLASS_ATTR_FLOAT_ARRAY(c,"amplitudes",0,t_oscil,a_amplitudes,
    OSCIL_MAX_HARMS);
CLASS_ATTR_ACCESSORS(c, "amplitudes", a_amplitudes_get,
    a_amplitudes_set);
CLASS_ATTR_LABEL(c, "amplitudes", 0, "Harmonic Weightings");
```

Figure 10.16 Defining the amplitudes attribute in the initialization routine

```
t_max_err a_amplitudes_set(t_oscil *x, void *attr, long ac,
    t_atom *av)
{
    int i;
    t_atom *rv = NULL;
    if (ac&&av) {
          for(i = 0; i < OSCIL_MAX_HARMS; i++){
                x->a_amplitudes[i] = atom_getfloatarg(i, ac, av);
          }
    }
    object_method_sym((t_object *)x, gensym("waveform"),
          gensym("Additive"), rv);
    return MAX_ERR_NONE;
}
```

Figure 10.17 Allocating memory for the amplitudes attribute

```
t_max_err a_amplitudes_set(t_oscil *x, void *attr, long ac,
    t_atom *av)
{
    int i;
    t_atom *rv = NULL;
    if (ac&&av) {
          for(i = 0; i < OSCIL_MAX_HARMS; i++){
                x->a_amplitudes[i] = atom_getfloatarg(i, ac, av);
          }
    }
    object_method_sym((t_object *)x, gensym("waveform"),
          gensym("Additive"), rv);
    return MAX_ERR_NONE;
}
```

Figure 10.18 The `amplitudes` setter method

```
t_max_err a_amplitudes_get(t_oscil *x, void *attr, long *ac,
   t_atom **av)
{
   int i;

   if (!((*ac)&&(*av))) {
         *ac = OSCIL_MAX_HARMS;
         if (!(*av = (t_atom *)sysmem_newptr(sizeof(t_atom) *
               (*ac))))
         {
               *ac = 0;
               return MAX_ERR_OUT_OF_MEM;
         }
   }
   for(i = 0; i < OSCIL_MAX_HARMS; i++){
         atom_setfloat(*av + i,x->a_amplitudes[i]);
   }
   return MAX_ERR_NONE;
}
```

Figure 10.19 The `amplitudes` getter method

```
void oscil_additive(t_oscil *x)
{
   int i;
   x->harmonic_count = 0;
   for(i = 0; i < OSCIL_MAX_HARMS; i++){
         x->amplitudes[i] = x->a_amplitudes[i];
         if(x->a_amplitudes[i]){
               x->harmonic_count = i;
         }
   }
   oscil_build_waveform(x);
}
```

Figure 10.20 The `oscil_additive()` routine

```
CLASS_ATTR_ORDER(c, "frequency", 0, "1");
CLASS_ATTR_ORDER(c, "waveform", 0, "2");
CLASS_ATTR_ORDER(c, "xfade", 0, "3");
CLASS_ATTR_ORDER(c, "amplitudes", 0, "4");
```

Figure 10.21 Defining the order of appearance for the attributes of *oscil~*

```
#include "stdio.h"
main()
{
  float x, y;
  y = x * z;
}
```

Figure 11.1 A defective C program with an easy-to-find bug

```
#include "stdio.h"
#include "stdlib.h"
main()
{
  float *mem1;
  float *mem2;
  int i;
  int len = 32768;

  mem1 = (float *) malloc(len * sizeof(float));
  mem1 = (float *) malloc(len * sizeof(float));
  for(i = 0; i < len; i++){
    mem1[i] = i;
  }
  for(i = 0; i < len; i++){
    mem2[i] = mem1[i] * i;
  }
}
```

Figure 11.3 A buggy program

```
 1 #include "stdio.h"
 2 #include "stdlib.h"
 3 #include "math.h"
 4 #define TWOPI 3.1415926535898
 5
 6 main()
 7 {
 8   float line[64];
 9   float sine[64];
10   buildsine(line,sine,64);
11 }
12
13 buildsine(float *line, float *sine, int length){
14   int i;
15   for(i = 0; i < length; i++){
16     line[1] = TWOPI / length;
17   }
18   for(i= 0; i < length; i++){
19     sine[i] = sin(line[i]);
20   }
21 }
```

Figure 11.4 A failed attempt to build and store a digital sine wave

```
main()
{
  int i;
  float line[64];
  float sine[64];
  buildsine(line,sine,16);
  for(i= 0; i < 16; i++){
    printf("%f\n",sine[i]);
  }
}
```

Figure 11.5 Rewriting the `main()` program to print the data generated by `buildsine()`

```
line[i] = TWOPI / length;
```

Figure 11.7 Fixing the typo on line 16

```
line[i] = TWOPI * i / length;
```

Figure 11.9 Incorporating the index variable into the calculation

```
#define TWOPI 3.1415926535898
```

Figure 11.11 Incorrectly defined constant for 2π

```
#include "m_pd.h"
static t_class *ramp_class;

typedef struct _ramp
{
    t_object obj;
    float x_f;
    long counter;
    long maximum;
} t_ramp;

void *ramp_new(void);
t_int *ramp_perform(t_int *w);
void ramp_dsp(t_ramp *x, t_signal **sp, short *count);

void ramp_tilde_setup (void)
{
    ramp_class = class_new(gensym("ramp~"), (t_newmethod)ramp_new,0,
            sizeof(t_ramp), 0,A_GIMME,0);
    t_class *c;
    class_addmethod(c, (t_method)ramp_dsp, gensym("dsp"), A_CANT, 0);
    CLASS_MAINSIGNALIN(c, t_ramp, x_f);
    post("ramp~: from \"Designing Audio Objects \" by Eric Lyon");
}

void *ramp_new(void)
{
    t_ramp *x = (t_ramp *)pd_new(ramp_class);
    inlet_new(&x->obj, &x->obj.ob_pd, gensym("signal"),
            gensym("signal"));
     outlet_new(&x->obj, gensym("signal"));
     outlet_new(&x->obj, gensym("signal"));
      x->maximum = 44100;
      x->counter = 0;
     return NULL;
}

t_int *ramp_perform(t_int *w)
{
    t_ramp *x = (t_ramp *) (w[1]);
    float *trigger = (t_float *)(w[2]);
    float *maxcount = (t_float *)(w[3]);
    float *out = (t_float *)(w[4]);
    int n = w[5];
    long maximum = x->maximum;
    long counter = x->counter;
    float invmax;
    for(i = 0; i < n; i++){
            if(trigger[i]){
                    counter = 0;
                    maximum = maxcount[i];
            }
            out[i] = counter * invmax;
            if(counter < maximum){
                    counter++;
            }
    }
    return w + 7;
}

void ramp_dsp(t_ramp *x, t_signal **sp, short *count)
{
    dsp_add(ramp_perform, 7, x, sp[0]->s_vec, sp[1]->s_vec, sp[0]->s_n);
}
```

Figure 11.13 The defective code for *ramp~*

```
int main(void)
{
   ramp_class = class_new("ramp~", (method)ramp_new,
               (method)dsp_free, sizeof(t_ramp), 0,0);
   t_class *c;
   class_addmethod(c, (method)ramp_dsp, "dsp", A_CANT, 0);
   class_addmethod(c, (method)ramp_assist, "assist", A_CANT, 0);
   class_dspinit(c);
   class_register(CLASS_BOX, c);
   post("ramp~ from \”Designing Audio Objects\” by Eric Lyon”);
   return 0;
}
```

Figure 11.19 The defective `main()` function for *ramp~* in Max/MSP

```
int count; // risky
int count = 0; // safe
```

Figure 11.20 Watch out for uninitialized variables

```
int trouble[32];
int i;

/* in this loop we go one address too far, ending with
   an illegal write to the array trouble[]. */

for(i = 0; i <= 32; i++){
   trouble[i] = i;
}
```

Figure 11.21 Bad behavior at the extremes

```
typedef struct {
    OPDS    h;
    MYFLT   *out, *in, *fco, *res, *max, *iskip;
    double  xnm1, y1nm1, y2nm1, y3nm1, y1n, y2n, y3n, y4n;
    MYFLT   maxint;
    int16   fcocod, rezcod;
} MOOGVCF;
```

Figure 12.2 The unit generator structure for *moogvcf*

```
static int moogvcf(CSOUND *csound, MOOGVCF *p)
{
    int n, nsmps = csound->ksmps;
    MYFLT *out, *in;
    double xn;
    MYFLT *fcoptr, *resptr;
    /* Fake initialisations to stop compiler warnings!! */
    double fco, res, kp=0.0, pp1d2=0.0, scale=0.0, k=0.0;
    double max = (double)p->maxint;
    double dmax = 1.0/max;
    double xnm1 = p->xnm1, y1nm1 = p->y1nm1,y2nm1 = p->y2nm1,
           y3nm1 = p->y3nm1;
    double y1n  = p->y1n, y2n = p->y2n, y3n = p->y3n, y4n = p->y4n;
    in      = p->in;
    out     = p->out;
    fcoptr  = p->fco;
    resptr  = p->res;
    fco     = (double)*fcoptr;
    res     = (double)*resptr;
  /* Only need to calculate once */
    if (UNLIKELY((p->rezcod==0) && (p->fcocod==0))) {
      double fcon;
      fcon  = 2.0*fco*(double)csound->onedsr; /* normalised frq. 0 to Nyq */
      kp    = 3.6*fcon-1.6*fcon*fcon-1.0;     /* Empirical tuning   */
      pp1d2 = (kp+1.0)*0.5;                   /* Timesaver          */
      scale = exp((1.0-pp1d2)*1.386249);      /* Scaling factor     */
      k     = res*scale;
    }
    for (n=0; n<nsmps; n++) {
      /* Handle a-rate modulation of fco & res. */
      if (p->fcocod) {
        fco = (double)fcoptr[n];
      }
      if (p->rezcod) {
        res = (double)resptr[n];
      }
      if ((p->rezcod!=0) || (p->fcocod!=0)) {
        double fcon;
        fcon  = 2.0*fco*(double)csound->onedsr; /* normalised frq. 0 to Nyquist */
        kp    = 3.6*fcon-1.6*fcon*fcon-1.0;     /* Empirical tuning */
        pp1d2 = (kp+1.0)*0.5;                   /* Timesaver */
        scale = exp((1.0-pp1d2)*1.386249);      /* Scaling factor */
        k     = res*scale;
      }
      xn = (double)in[n] * dmax;
      xn = xn - k * y4n; /* Inverted feed back for corner peaking */
      /* Four cascaded onepole filters (bilinear transform) */
      y1n   = (xn  + xnm1 ) * pp1d2 - kp * y1n;
      y2n   = (y1n + y1nm1) * pp1d2 - kp * y2n;
      y3n   = (y2n + y2nm1) * pp1d2 - kp * y3n;
      y4n   = (y3n + y3nm1) * pp1d2 - kp * y4n;
                                  /* Clipper band limited sigmoid */
      y4n   = y4n - y4n * y4n * y4n / 6.0;
      xnm1  = xn;       /* Update Xn-1  */
      y1nm1 = y1n;      /* Update Y1n-1 */
      y2nm1 = y2n;      /* Update Y2n-1 */
      y3nm1 = y3n;      /* Update Y3n-1 */
      out[n]   = (MYFLT)(y4n * max);
    }
    p->xnm1 = xnm1; p->y1nm1 = y1nm1; p->y2nm1 = y2nm1;
           p->y3nm1 = y3nm1;
    p->y1n  = y1n;  p->y2n  = y2n; p->y3n = y3n; p->y4n = y4n;
    return OK;
}
```

Figure 12.3 C code for the Csound *moogvcf* unit generator

```
if ((p->rezcod==0) && (p->fcocod==0))
```

Figure 12.4 The condition for computing new coefficients at the control rate

```
typedef struct _moogvcf
{
      t_pxobject obj;
      double  xnm1, y1nm1, y2nm1, y3nm1, y1n, y2n, y3n, y4n;
      double onedsr; // one divided by the sample rate
} t_moogvcf;
```

Figure 12.5 The object structure for a Max/MSP implementation of *moogvcf~*

```
void moogvcf_dsp(t_moogvcf *x, t_signal **sp, short *count)
{
    if(sp[0]->s_sr){
          x->onedsr = 1.0 / sp[0]->s_sr;
          dsp_add(moogvcf_perform, 6, x, sp[0]->s_vec,
            sp[1]->s_vec,sp[2]->s_vec, sp[3]->s_vec, sp[0]->s_n);
    }
}
```

Figure 12.6 Setting the value of `onedsr`

```
t_int *moogvcf_perform(t_int *w)
{
   t_moogvcf *x = (t_moogvcf *) (w[1]);
   float *input = (t_float *)(w[2]);
   float *frequency = (t_float *)(w[3]);
   float *resonance = (t_float *)(w[4]);
   float *output = (t_float *)(w[5]);
   int n = w[6];
   double fcon;
   double onedsr = x->onedsr;
   double kp=0.0, pp1d2=0.0, scale=0.0, k=0.0;
   double xn;
   double xnm1 = x->xnm1, y1nm1 = x->y1nm1,
         y2nm1 = x->y2nm1, y3nm1 = x->y3nm1;
   double y1n  = x->y1n, y2n = x->y2n, y3n = x->y3n, y4n = x->y4n;

   while(n--){
         fcon  = 2.0 * *frequency++ * onedsr;
               /* normalised frq. 0 to Nyquist */
         kp    = 3.6*fcon-1.6*fcon*fcon-1.0;
               /* Empirical tuning */
         pp1d2 = (kp+1.0)*0.5;
               /* Timesaver */
         scale = exp((1.0-pp1d2)*1.386249);
               /* Scaling factor */
         k     = *resonance++ * scale;
         xn = *input++;
         xn = xn - k * y4n;
               /* Inverted feed back for corner peaking */
         y1n   = (xn  + xnm1 ) * pp1d2 - kp * y1n;
         y2n   = (y1n + y1nm1) * pp1d2 - kp * y2n;
         y3n   = (y2n + y2nm1) * pp1d2 - kp * y3n;
         y4n   = (y3n + y3nm1) * pp1d2 - kp * y4n;
         y4n   = y4n - y4n * y4n * y4n / 6.0;
         xnm1  = xn;       /* Update Xn-1  */
         y1nm1 = y1n;      /* Update Y1n-1 */
         y2nm1 = y2n;      /* Update Y2n-1 */
         y3nm1 = y3n;      /* Update Y3n-1 */
         *output++   =  y4n;
   }
    x->xnm1 = xnm1; x->y1nm1 = y1nm1;
         x->y2nm1 = y2nm1; x->y3nm1 = y3nm1;
    x->y1n  = y1n;  x->y2n  = y2n; x->y3n = y3n; x->y4n = y4n;
   return w + 7;
}
```

Figure 12.7 The perform routine for *moogvcf~*

```
fcon  = 1.78179 * *frequency++ * onedsr;
```

Figure 12.9 Adjusting the tuning of *moogvcf~*

```
var points = 8192;
outlets = 2;
function linefunc1(segs){
    var seglen = points / segs;
    var firstpoint = x1 = (Math.random() * 2) - 1;
    var x2 = (Math.random() * 2) - 1;
    var i,j;
    var sample, frac;
    for(i = 0; i < segs; i++){
        for(j = 0; j < seglen; j++){
            sample = x1 + ((j/seglen) * (x2 - x1));
            outlet(1, (i*seglen)+j);
            outlet(0, sample);
        }
        x1 = x2;
        if(i == segs - 2){
            x2 = firstpoint;
        } else {
            x2 = (Math.random() * 2) - 1;
        }
    }
}
```

Figure 13.1 JavaScript code to initialize a stochastic waveform

```
var length = 16;
var sequence = new Array(length);

randomize(length);

function getfreq(index)
{
   outlet(0, sequence[index]);
}

function randomize(length)
{
   var i;
   for(i = 0; i < length; i++){
         sequence[i] = 60 + (Math.random() * 400);
   }
}
```

Figure 14.5 The live-sequencer JavaScript code

```
void dsp_add(t_perfroutine f, int n, ...)
{
    int newsize = dsp_chainsize + n+1, i;
    va_list ap;

    dsp_chain = t_resizebytes(dsp_chain,
         dsp_chainsize * sizeof (t_int), newsize * sizeof
   (t_int));
    dsp_chain[dsp_chainsize-1] = (t_int)f;
    va_start(ap, n);
    for (i = 0; i < n; i++)
        dsp_chain[dsp_chainsize + i] = va_arg(ap, t_int);
    va_end(ap);
    dsp_chain[newsize-1] = (t_int)dsp_done;
    dsp_chainsize = newsize;
}
```

Figure 15.2 The function `dsp_add()` from the Pd source code